UNISYS

V Series
COBOL ANSI-74
Programming Reference Manual
Volume 1: Basic Implementation

Previous Title: V Series COBOL ANSI-74 Compiler Programming Reference Manual

Release 3.1

April 1990
Distribution Code SD
Printed in the United States
4127 4945-000

Priced Item

UNISYS

V Series
COBOL ANSI-74
Programming Reference Manual
Volume 1: Basic Implementation

Copyright © 1990 Unisys Corporation
All rights reserved.
Unisys is a registered trademark of Unisys Corporation

Release 3.1

April 1990
Distribution Code SD
Printed in U S America

Priced Item

4127 4945-000

The names, places, and/or events used in this publication are not intended to correspond to any individual, group, or association existing, living, or otherwise. Any similarity or likeness of the names, places, and/or events with the names of any individual, living or otherwise, or that of any group or association is purely coincidental and unintentional.

NO WARRANTIES OF ANY NATURE ARE EXTENDED BY THE DOCUMENT. Any product and related material disclosed herein are only furnished pursuant and subject to the terms and conditions of a duly executed Program Product License or Agreement to purchase or lease equipment. The only warranties made by Unisys, if any, with respect to the products described in this document are set forth in such License or Agreement. Unisys cannot accept any financial or other responsibility that may be the result of your use of the information in this document or software material, including direct, indirect, special or consequential damages.

You should be very careful to ensure that the use of this information and/or software material complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions. COBOL is an industry language and is not the property of any company or group of companies, or of any organization or group of organizations.

No warranty, expressed or implied, is made by any contributor or by the CODASYL Programming Language Committee as to the accuracy and functioning of the programming system and language. Moreover, no responsibility is assumed by any contributor, or by the committee, in connection therewith.

The authors and copyright holders of the copyrighted material used herein have specifically authorized the use of this material, in whole or in part, in the COBOL specifications. These authors or copyright holders are the following:

FLOW-MATIC (Trademark of Sperry Rand Corporation), Programming for the UNIVAC(R) I and II, Data Automation Systems copyrighted 1958, 1959, by Sperry Rand Corporation; IBM Commercial Translator Form No. F 28-8013, copyrighted 1959 by IBM; FACT, DSI 27A5260-2760, copyrighted 1960 by Minneapolis-Honeywell

Such authorization extends to the reproduction and use of COBOL specifications in programming manuals or similar publications.

Correspondence regarding this publication should be forwarded to Unisys Corporation either by using the Business Reply Mail form at the back of this manual or by addressing remarks directly to Unisys Corporation, Technical Publications, 460 Sierra Madre Villa, Pasadena, CA 91107, U.S.A.

Page Status

Page	Issue
iii	Original
iv	Blank
v through xxv	Original
xxvi	Blank
xxvii through xxix	Original
xxx	Blank
1–1 through 1–12	Original
2–1 through 2–14	Original
3–1 through 3–3	Original
3–4	Blank
4–1 through 4–33	Original
4–34	Blank
5–1 through 5–73	Original
5–74	Blank
6–1 through 6–176	Original
7–1 through 7–28	Original
8–1 through 8–24	Original
9–1 through 9–24	Original
10–1 through 10–3	Original
10–4	Blank
A–1 through A–4	Original
B–1 through B–18	Original
C–1 through C–8	Original
D–1 through D–3	Original
D–4	Blank
E–1 through E–17	Original
E–18	Blank
F–1 through F–58	Original
Glossary–1 through Glossary–25	Original
Glossary–26	Blank
Index–1 through Index–84	Original

About This Manual

Purpose

This manual is the first volume of a two-volume set. This volume describes the syntax and general usage of Unisys V Series COBOL ANSI-74. *V Series COBOL74 Programming Reference Manual, Volume 2: Product Interfaces* provides information on using Unisys V Series COBOL ANSI-74 to write application programs that interface with Data Management System II (DMSII), Screen Design Facility (SDF), and Reader-Sorters.

Scope

This manual describes the Unisys V Series COBOL ANSI-74 language. This includes American National Standard Institute (ANSI) COBOL 1974, as well as Unisys V Series extensions to the COBOL ANSI-74 Standard. This document is intended not as a training device, but only as a reference guide.

Audience

The primary audience of this manual includes experienced programmers who use COBOL to create programs or who need to understand programs written in this language. A secondary audience includes programmers attempting to learn COBOL. However, the document is not structured as a training device for such an audience.

Prerequisites

Programmers using this document should be familiar with the general concepts of COBOL programming.

How to Use This Manual

This volume of the Programming Reference Manual is a reference document that should be used with Volume 2.

Review the entire manual to familiarize yourself with its contents, and then use this document as necessary to guide you when writing and maintaining V Series COBOL ANSI-74 programs.

To find information about a specific standard COBOL statement, refer to Sections 3, 4, 5, or 6.

To find information about Unisys extensions to ANSI-74 COBOL, refer to text printed in blue ink.

To find information about coding formats, refer to Sections 1 and 2.

To find information about file attributes, refer to Section 7.

To find information about DMSII, refer to Sections 1 through 4 of Volume 2.

To find information about the COBOL interface to the Screen Design Facility (SDF), refer to Section 5 of Volume 2.

To find information about Reader-Sorter interface with COBOL, refer to Section 6 of Volume 2.

Other reference material is included throughout this manual.

Organization

This volume of the Programming Reference Manual is divided into ten sections and six appendices, as described in the following paragraphs.

Section 1: Overview and Format

This section contains a brief overview of Unisys V Series COBOL ANSI-74, including a brief description of the characteristics associated with this language. This section can help you become familiar with the concepts and terminology that are basic to the language.

Section 2: Language Elements

This section presents detailed and specific rules to be followed when you create a COBOL program.

Section 3: Identification Division

This section describes the function, format, and elements of the Identification Division, which identifies the COBOL source program.

Section 4: Environment Division

This section describes the second division of a COBOL source program. All aspects of a data processing problem that depend on the physical characteristics of a specific computer are expressed in this section.

Section 5: Data Division

This section describes the third division of a COBOL source program. This division contains the data that the COBOL object program is to accept as input, manipulate, create, or produce as output.

Section 6: Procedure Division

This section describes the final division of a COBOL source program, the Procedure Division. Specific statement formats and detailed descriptions of COBOL instructions in this section help you solve given problems.

Section 7: File Attributes

This section describes in detail all the functions associated with file attributes.

Section 8: File Handling

This section describes file handling, the physical aspects of files, file organization, and file access methods.

Section 9: Compiler Operation

This section describes how to compile a COBOL program and the use of compiler intrinsics and dollar options.

Section 10: COBOL Binding

This section describes the advantages, considerations, and requirements of COBOL program binding.

Appendix A: Reserved Words

This appendix contains a list of reserved words for COBOL ANSI-74 and Unisys V Series COBOL ANSI-74 extensions.

Appendix B: Warning and Error Messages

This appendix lists the fatal and nonfatal error messages generated by the COBOL compiler, along with a brief explanation of the cause of each message.

Appendix C: EBCDIC, ASCII, and BCL Reference Tables

This appendix contains table specifications of EBCDIC, ASCII, and BCL code as implemented by Unisys for V Series processors.

Appendix D: Debugging Aids

This appendix describes the layout of object program memory and provides other information on debugging COBOL programs.

Appendix E: Program Examples

This appendix contains sample COBOL programs.

Appendix F: COBOL Conversion Aid

This appendix presents detailed information about the functions and usage of the COBOL Conversion Aid program.

Glossary

This contains a list of glossary terms and their definitions.

Results

After using this document, you will be more familiar with the Unisys V Series implementation of the COBOL ANSI-74 programming language.

You will be able to use this manual as a reference tool to find answers to specific questions about the COBOL language and to interpret syntax of existing COBOL programs.

Related Product Information

The following documents may be useful in understanding and implementing the specifications that are defined in this document:

American National Standard Programming Language COBOL ANSI X3.23–1974

This standard provides specifications for both the form and interpretation of programs written in the COBOL language.

Federal Information Processing Standards Publication, FIBS PUB 21–1, December 1, 1975

This publication describes the COBOL language standards for use by all Federal departments and agencies.

V Series COBOL74 Programming Reference Manual, Volume 2: Product Interfaces (4127 4952–000)

This document provides detailed information on using V Series COBOL ANSI-74 to write application programs that interface with Data Management System II (DMSII), Screen Design Facility (SDF), and Reader-Sorters.

V Series DMSII Operations Reference Manual, Volume 1: Database Creation (5031412-001).

This manual includes step-by-step instructions for creating a database in the Data Structure and Definition Language (DASDL). It also includes descriptions of the structures that a programmer uses to define each database, examples of the structures, examples of databases, information on remaps and logical databases, and instructions for creating and compiling a DASDL source file.

V Series DMSII Operations Reference Manual, Volume 2: Database Administration (5031404-001).

This manual includes information about file naming conventions, related commands, updating and reorganizing a database. In addition, this manual contains procedures for audit and recovery of databases, instructions for dumping and loading databases, exception conditions for user programs, and error messages and system messages for operators.

V Series DMSII Operations Reference Manual, Volume 3: System Utilities (5031420-001)

This manual provides step-by-step instructions for each of the DMSII utilities. In addition, this manual contains all the system messages and error messages that pertain to DMSUTL.

V Series Systems Operations Guide, Volume 2: System Commands (4127 4994-000)

This document explains the purpose and syntax of the system commands and control instructions.

V Series Systems Operations Guide, Volume 3: System Utilities (4127 0000-000)

This document explains the system utility programs.

V Series Systems Operations Guide, Volume 4: System Messages and Recovery (4127 0018-000)

This document describes the system output messages.

V Series CANDE Installation and Operations Guide (5030141)

This document describes the V Series Command and Edit (CANDE) text editor.

V Series System Security Installation and Operations Reference Manual (4127 4820-000)

This document describes the V Series System Security features.

B 2000/B 3000/B 4000/V Series Screen Design Facility (SDF) User's Guide (5024714)

This document describes the concepts, features, and operations of the Screen Design Facility (SDF) to create screens in application programs.

V Series MCP/VS Programming Reference Manual (4127 0026-000)

This document provides detailed information on the MCP that is of interest to system and application programmers.

V Series Intelligent Workstation Editor II (IWEII) Installation and Operations Guide (5031438)

This document describes the concepts, features, and operations of the IWEII application program.

BNA Architectual Description Reference Manual, Volume 1 (1132172)

This document provides a functional description of BNA Version 1.

Acknowledgment

Any organization interested in reproducing the COBOL report and specifications in whole or in part, using ideas taken from this document as the basis for an instruction manual or for any other purpose, is free to do so. However, all such organizations are requested to reproduce this section in its entirety as part of the preface to the document. Those using a short passage, as in a book review, are requested to mention "COBOL" in acknowledgment of the source, but need not quote this entire section.

COBOL is an industry language and is not the property of any company or group of companies, or of any organization or group of organizations.

No warranty, expressed or implied, is made by any contributor or by the CODASYL Programming Language Committee as to the accuracy and functioning of the programming system and language. Moreover, no responsibility is assumed by any contributor, or by the committee, in connection therewith.

These authors and copyright holders of the copyrighted material used herein have specifically authorized the use of this material in whole or in part, in the COBOL specifications.

FLOW-MATIC (Trademark of Sperry Rand Corporation), Programming for the UNIVAC (R) I and II, Data Automation Systems copyrighted 1958, 1959, by Sperry Rand Corporation

IBM Commercial Translator Form No. F 28–8013, copyrighted 1959 by IBM

FACT, DSI 27A5260–2760, copyrighted 1960 by Minneapolis-Honeywell

This authorization extends to the reproduction and use of COBOL specifications in programming manuals or similar publications.

Contents

About This Manual v

Section 1. Overview and Format

Overview 1–1
Source Program 1–2
Structure 1–2
Compiler Problems 1–2
Format 1–3
Notation Used in Formats 1–3
Words 1–3
Level Numbers 1–4
Brackets and Braces 1–4
Ellipsis 1–5
Punctuation 1–5
Special Characters 1–5
Coding Form 1–6
Field Definitions 1–6
1–6 Sequence Area 1–8
7 Indicator Area 1–8
Comment Line 1–8
Continuation Line 1–9
Debugging Line 1–10
Compiler Control Options 1–11
8–11 Area A 1–11
12–72 Area B 1–11
73–80 Identification 1–11
Blank Lines 1–12
Sample Coding Form Usage 1–12

Section 2. Language Elements

Character Set 2–1
Language Structure 2–2
Separators 2–2
Character Strings 2–3
COBOL Words 2–3
User-Defined Words 2–4
System Names 2–6
Reserved Words 2–7
Literals 2–11

Nonnumeric Literals 2–11
Numeric Literals 2–12
Hexadecimal Literals 2–13
Numeric Data Items 2–14
Alphanumeric Data Items 2–14
PICTURE Character Strings 2–14
Comment Entries 2–14

Section 3. Identification Division

Structure 3–1
Coding Example 3–2
PROGRAM-ID Paragraph 3–2
DATE-COMPILED Paragraph 3–3

Section 4. Environment Division

Structure 4–1
Configuration Section 4–2
Source-Computer Paragraph 4–2
Computer-Name Clause 4–2
WITH DEBUGGING MODE Clause 4–2
Object-Computer Paragraph 4–3
MEMORY SIZE Clause 4–3
STACK SIZE Clause 4–3
PROGRAM COLLATING SEQUENCE Clause ... 4–4
SEGMENT-LIMIT Clause 4–4
Special-Names Paragraph 4–5
CHANNEL Clause 4–6
ODT Clause 4–6
Switch Clause 4–6
Alphabet-Name Clause 4–7
Syntax Rules 4–8
General Rules 4–8
CURRENCY SIGN Clause 4–9
DECIMAL POINT IS COMMA Clause 4–9
Input-Output Section 4–10
File-Control Paragraph 4–10
SELECT Clause 4–13
OPTIONAL Phrase 4–14
ASSIGN Clause 4–14
Formats 2 and 3 4–17
Format 3 4–17
Format 4 4–17
Format 5 4–18
RESERVE Clause 4–18
HIGH MEMORY Clause 4–18
ORGANIZATION Clause 4–19
ACCESS MODE Clause 4–19
Sequential Files 4–19

Relative Files 4–20
Indexed Files 4–20
RELATIVE KEY Clause 4–20
RECORD KEY and ALTERNATE RECORD KEY Clauses 4–21
ACTUAL KEY Clause 4–22
FILE STATUS Clause 4–22
Sequential Files 4–22
Relative Files 4–24
Indexed Files 4–26
I-O-Control Paragraph 4–31
SAME Clause 4–31
MULTIPLE FILE Clause 4–33
Coding Example 4–33

Section 5. Data Division

General 5–1
Structure 5–1
Records 5–2
Levels 5–2
Level Numbers 5–3
Data 5–6
Data Classification 5–6
Unique Data References 5–6
Qualification 5–7
Data Alignment 5–8
Tables 5–9
Table Definition 5–9
Table Handling 5–10
Subscripting 5–11
Indexing 5–12
Subscript and Index Validation 5–13
Indexing Versus Subscripting 5–14
File Section 5–16
File Description Entry 5–16
Level Indicators (FD,SD) 5–19
BLOCK CONTAINS Clause 5–20
CODE-SET Clause 5–22
COMMON Clause 5–23
DATA RECORDS Clause 5–24
LABEL RECORDS Clause 5–25
LINAGE Clause 5–27
Rules of Use 5–28
Practical Considerations 5–29
RECORD CONTAINS Clause 5–30
VALUE OF Clause 5–32
Record Description 5–33
Data Description Entry 5–33
Clauses in the Data Description Entry 5–36
BLANK WHEN ZERO Clause 5–37

COMMON Clause 5–38
Condition-Names 5–39
Data Name or FILLER Clause 5–41
HIGH MEMORY Clause 5–42
JUSTIFIED Clause 5–44
Level Number 5–45
OCCURS Clause 5–46
PICTURE Clause 5–49
REDEFINES Clause 5–60
RENAMES Clause 5–62
SIGN Clause 5–64
SYNCHRONIZED Clause 5–66
USAGE Clause 5–67
VALUE Clause 5–69
Working-Storage Section 5–71
Noncontiguous Working-Storage 5–71
Working-Storage Records 5–71
Initial Values 5–71
Coding Example 5–71
Linkage Section 5–73
Inter-Program Communication 5–73

Section 6. Procedure Division

General 6–1
Structure 6–2
Qualification 6–4
Segmentation 6–5
Fixed Segments 6–5
Independent Segments 6–6
Segmentation Logic 6–6
Segment-Numbers 6–7
Overlay Areas 6–7
Statements and Sentences 6–10
Conditional Statements and Sentences 6–10
Compiler-Directing Statements and Sentences 6–10
Imperative Statements and Sentences 6–11
Arithmetic Expression 6–12
Arithmetic Operators 6–12
Formation and Evaluation of Arithmetic Expressions . 6–13
Intermediate Data Item 6–14
Conditional Expressions 6–15
Simple Condition 6–15
Relation Condition 6–15
Comparison of Numeric Operands 6–16
Comparison of Nonnumeric Operands ... 6–16
Comparisons Involving Index-Names and/or Index Data Items 6–17
Comparisons Involving Mnemonic File Attributes 6–17
Class Condition 6–18

Condition-Name Condition 6–19
Switch-Status Condition 6–19
Sign Condition 6–20
Complex Conditions 6–20
Negated Simple Conditions 6–21
Combined and Negated Combined Conditions . 6–21
Abbreviated Combined Relation Conditions 6–23
Condition Evaluation Rules 6–24
Numeric Functions: Offset Function 6–25
Common Phrases 6–26
ROUNDED Phrase 6–26
SIZE ERROR Phrase 6–26
CORRESPONDING Phrase 6–27
General Rules for Statement Formats 6–28
Arithmetic Statements 6–28
Overlapping Operands 6–28
Multiple Results in Arithmetic Statements 6–28
Incompatible Data 6–29
Specific Formats 6–29
ACCEPT Statement 6–30
Format 1 6–30
Format 2 6–31
Format 3 6–32
ADD Statement 6–33
ALTER Statement 6–35
CALL Statement 6–36
CALL INTRINSIC Statement 6–39
Format 1 6–42
Format 2 6–43
Format 3 6–43
Format 4 6–45
Format 5 6–45
Format 6 6–46
Format 7 6–47
Format 8 6–47
Format 9 6–49
Format 10 6–50
Format 11 6–52
CALL MODULE Statement 6–53
CALL SYSTEM Statement 6–55
Format 1 6–56
Format 2 6–56
Format 3 6–57
CANCEL Statement 6–58
CHANGE Statement 6–59
CLOSE Statement 6–61
COMPUTE Statement 6–65
COPY Statement 6–67
REPLACING Clause 6–68
Library 6–71
Creating Libraries 6–71
Using Libraries 6–72

DELETE Statement 6–73
DISPLAY Statement 6–74
DIVIDE Statement 6–76
EXIT Statement 6–79
Format 1 6–80
Format 2 6–80
Format 3 6–80
GO Statement 6–81
IF Statement 6–82
INSPECT Statement 6–83
Tallying 6–87
Replacing 6–87
MERGE Statement 6–89
MOVE Statement 6–93
Illegal Moves 6–95
MOVE Results 6–95
MULTIPLY Statement 6–97
OPEN Statement 6–98
PERFORM Statement 6–103
PERFORM Logic Cycle 6–106
Format 1 6–106
Format 2 6–107
Format 3 6–107
Format 4 6–107
READ Statement 6–112
Format 1 6–112
Sequential Files 6–113
Relative Files 6–113
Indexed Files 6–113
Format 1 Details 6–113
Format 2 6–115
Sequential Files 6–116
Relative Files 6–116
Indexed Files 6–116
Port Files 6–117
DCP Files 6–117
RECEIVE Statement 6–118
Format 1 6–118
Format 2 6–119
RELEASE Statement 6–121
RETURN Statement 6–122
REWRITE Statement 6–124
SEARCH Statement 6–126
Format 1 6–127
Format 2 6–130
SEEK Statement 6–131
SEND Statement 6–132
Format 1 6–132
Format 2 6–133
Storage Queue Description 6–134
SET Statement 6–136
SORT Statement 6–138

START Statement 6–144
STOP Statement 6–146
Clearing of IX1 at STOP RUN (COMPILE) 6–147
STRING Statement 6–148
SUBTRACT Statement 6–151
UNSTRING Statement 6–153
Rules for Data Transfer 6–156
USE Statement 6–158
Format 1 6–159
Monitoring Data Items 6–160
Monitoring Files 6–161
Monitoring Procedures 6–161
DEBUG-ITEM 6–161
Format 2 6–164
Format 3 6–165
Format 4 6–165
WAIT Statement 6–167
WRITE Statement 6–170
DCP Files 6–176

Section 7. File Attributes

General 7–1
Categories 7–1
Alphanumeric Attributes 7–2
Numeric Attributes 7–2
Mnemonic Attributes 7–2
References to File Attributes 7–3
Attributes and Values 7–5
ACCESSMODE 7–8
AREAS 7–8
AREASIZE 7–8
BACKUPKIND 7–9
BACKUPPERMITTED 7–9
BLOCK 7–9
BLOCKSIZE 7–9
BLOCKSTRUCTURE 7–10
BUFFERS 7–10
CREATIONDATE 7–10
CURRENTBLOCK 7–10
CYLINDERMODE 7–11
DENSITY 7–11
DIRECTION 7–11
EXTMODE 7–12
FAMILYNAME 7–12
FILENAME 7–12
FORCEIO 7–13
FORMID 7–13
HOSTNAME 7–13
KIND 7–14
Tape Considerations 7–14

LABELKIND 7–14
LABELSIZE 7–15
LASTRECORD 7–15
MAXRECSIZE 7–15
MYUSE 7–16
OPEN 7–16
OPTIONAL 7–16
PARITY 7–16
PROTECTION 7–16
SAVEFACTOR 7–17
SECURITYFAMILY 7–17
SECURITYGUARD 7–17
SECURITYTYPE 7–17
SECURITYUSE 7–18
SENSITIVEDATA 7–18
SERIALNO 7–18
SHARED 7–18
SINGLEUNIT 7–19
SUBSYSTEM 7–19
SYNCHRONIZE 7–19
UNIQUENAME 7–20
UNITNO 7–20
VOLUMEINDEX 7–20
Task Attributes 7–21
ACCUMIOTIME 7–21
ACCUMPROCTIME 7–21
MIXNUMBER 7–21
PRIMARYSTORAGE 7–21
Port and Subport File Attributes 7–22
BLOCKSTRUCTURE 7–24
CENSUS 7–24
CHANGEDSUBFILE 7–24
CHANGEEVENT 7–24
COMPRESSION 7–24
COMPRESSIONPOSSIBLE 7–24
CURRENTRECORD 7–25
DIALOGPROTOCOLLEVEL 7–25
FILESTATE 7–25
HOSTNAME 7–25
INPUTEVENT 7–25
INTNAME 7–26
LASTSUBFILE 7–26
MAXCENSUS 7–26
MAXRECSIZE 7–26
MAXSUBFILES 7–26
MYHOSTNAME 7–26
MYNAME 7–26
OUTPUTEVENT 7–27
SUBFILEERROR 7–27
TITLE 7–27
YOURNAME 7–27
YOURUSERCODE 7–28

Attribute's Effect on System Performance 7–28

Section 8. File Handling

General 8–1
Physical Aspects of a File 8–1
File Concepts 8–1
File Organization and Access Methods 8–2
Sequential Organization 8–3
Relative Organization 8–4
Considerations for Implementing Relative Files 8–5
Indexed Organization 8–7
Shared Indexed Files 8–8
Considerations for Implementing Indexed Files 8–9
Considerations for Implementing Nonshared and Shared Files 8–9
Implementation Considerations for Nonshared Files . 8–11
Implementation Considerations for Shared Files 8–13
Indexed File Buffers 8–16
Nonshared vs. Shared Indexed File Buffer Considerations 8–16
Considerations for Nonshared Indexed File Buffers .. 8–16
Additional Buffer Considerations 8–18
Indexed File Recovery 8–18
Indexed File Error Indications 8–19
Shared File Functions 8–23
SELECT Clause 8–23
File Attribute SHARED 8–23
DECLARATIVES Section 8–24
Exception-Handling Procedures for Shared Files 8–24
Port Files 8–24

Section 9. Compiler Operation

General 9–1
Compilation File 9–1
COMPILE Statement 9–2
Label Equation Statement 9–6
Source Program 9–7
Input Files 9–7
Library Files 9–7
Compiler Intrinsics 9–8
Dollar Options 9–9
ANSIFILE 9–11
CHECK 9–11
CLEAR 9–11
CODE 9–12
CORRECTOK 9–12
CORRECTSUPR 9–12
DELETE 9–12

DMSICM ["file-id"] [device] 9–12
DOUBLE 9–12
ERRORLIMIT = nnn 9–13
FEDLEVEL [=] n 9–13
ICM 9–14
ICMCODE 9–15
ICMFAMILY "multi-file-id" 9–15
ICMFILE "file-id" 9–15
LEV2IPC 9–15
LIBR 9–16
LIBRFAMILY "multi-file-id" 9–16
LIST 9–16
LISTDELETED 9–16
LISTOMITTED 9–16
LISTP 9–17
LIST$ 9–17
MAIN 9–17
MAP 9–17
MCS 9–17
MERGE 9–17
NEW 9–18
NEWID "literal" 9–19
NOBOUNDS 9–19
NORETURN 9–19
OMIT 9–19
OPTIMIZE 9–19
PAGE 9–19
PAGESIZE = nn 9–19
POP 9–20
SAVECP 9–20
SAVERCD 9–20
SEQ [nnnnnn [+nnnnnn]] 9–20
SEQCHECK 9–20
SEQUENCE 9–20
SUMMARY 9–20
TADS 9–20
TITLE "literal" 9–21
VOID nnnnnn 9–21
WARNFATAL 9–21
WARNSUPR 9–21
XREF 9–21
Program Output 9–22
Object Code 9–22
Output Listing 9–22
Run-Time Errors 9–23
Time-of-Compile 9–23
Sizes and Limits 9–24

Section 10. COBOL Binding

Appendix A. Reserved Words

Appendix B. Warning and Error Messages

Fatal Errors B–1
Compiler Stack Overflow B–1
Compiler Address Error B–1
NIL Pointer Dereference B–1
Uninitialized Pointer Dereference B–2
Nonfatal Errors B–2
Environment Division Error Messages B–2
Data Division Error Messages B–3
Procedure Division Error Messages B–8
Syntax Error Messages B–12
Code Generation Phase Error Messages B–15
DMSII Error Messages B–16
Federal Level Warnings B–17
Compiler Error Messages B–17
Binding Phase Error Messages B–18

Appendix C. EBCDIC, ASCII, and BCL Reference Tables

Description of EBCDIC Codes C–5
Control and Special Codes C–5
Other Codes C–7

Appendix D. Debugging Aids

Object Program Memory Layout D–1
Reserved Memory D–1
Segment Dictionary D–1
Resident Segment D–2
Overlayed Segments D–2
High Data D–2
Stack D–2
Object Code File D–2
Debugging D–2

Appendix E. Program Examples

Appendix F. COBOL Conversion Aid

General Description F–1
Restrictions F–1
Control Statements F–2
File Equate (Label Equate) Information F–2

Use of the & Option Control Record F–2
RESEQ Option F–4
NORESEQ Option F–4
OBJECT-COMPUTER Name Option F–4
SWITCH Option F–5
COPY Option F–5
Examples of & Option Control Records F–6
Error Conditions With & Option Control Records F–6
What the COBOL Conversion Aid Does F–7
Identification Division F–7
Environment Division F–7
Configuration Section F–8
SOURCE-COMPUTER Paragraph F–8
OBJECT-COMPUTER Paragraph F–8
SPECIAL-NAMES Paragraph F–10
INPUT-OUTPUT SECTION F–11
FILE-CONTROL Paragraph F–11
I-O-CONTROL Paragraph F–15
Data Division F–15
File Section F–16
Working-Storage Section F–19
Abbreviations F–19
PICTURE Clause F–20
VALUE Clause F–20
OCCURS Clause F–21
REDEFINES Clause F–22
RENAMES Clause F–22
Procedure Division F–23
ACCEPT and DISPLAY Verbs F–23
ADD Verb F–23
ALTER Verb F–24
CLOSE Verb F–24
DIVIDE Verb F–25
ELSE/OTHERWISE Options F–26
ENTER Construct F–26
EXAMINE Format F–26
EQUALS Verb F–28
FILL Statement F–28
IF Statement F–29
INTERROGATE Statement F–29
Literals F–30
OPEN Statement F–30
READ Statement F–30
SEARCH Statement F–31
Section Headers F–32
SEEK Statement F–32
SORT Statement F–32
Special Registers F–32
TODAYS-DATE Register F–33
DATE Register F–33
TIME Register F–33

CORE-SIZE Register F–34
STOP Statement F–34
Switches (Internal Program Switches) F–34
TRACE Statement F–34
UNLOCK Statement F–35
USE Statement F–35
WRITE Statement F–36
ZIP Statement F–36
All Divisions F–36
BCL Plus Sign F–37
NOTE Statement F–37
New Reserved Words F–37
Separators F–40
Figurative Constants F–40
Blank Lines F–40
COPY Statement F–41
Periods F–41
Reference Format F–41
Data comm Constructs F–41
ACCEPT F–41
CLOSE F–42
DISABLE F–42
DISPLAY F–42
ENABLE F–43
FILL F–43
INTERROGATE F–43
OPEN F–43
READ F–44
WAIT F–44
WRITE F–44
WRITE-READ F–45
WRITE-READ-TRANS F–45
WRITE-TRANS-READ F–45
Reader-Sorter and Lister Constructs F–46
CONTROL - Reader-sorter F–46
OPEN - Reader-sorter F–46
READ - Reader-sorter F–46
SELECT - Reader-sorter F–47
USE - Reader-sorter F–47
CONTROL - Lister F–47
ENABLE - Lister F–48
WRITE - Lister F–48
Exception Conditions F–49
& Record Errors F–49
Input File Validation F–50
Expansion of Source Coding F–50
File Names F–50
Sequence Numbers F–50
Recognizing Ending Quote F–51
Record Names F–51
Equated Hardware Names F–51

Output From The COBOL Conversion Aid F–52
Blocking of Files F–52
Output Listing F–52
List of Messages F–53
Source Code Summary F–58

Glossary 1

Index 1

Figures

1–1. COBOL Coding Form and Reference Format 1–7

3–1. Structure of the Identification Division 3–1

4–1. Structure of the Environment Division 4–1

5–1. Structure of the Data Division 5–2
5–2. Example of a Record Format 5–4
5–3. PICTURE Character Precedence Chart 5–59

6–1. Example of a Program with Multiple Overlay Areas 6–9
6–2. Flowchart for the VARYING Phrase of a PERFORM Statement Having One Condition. 6–109
6–3. Flowchart for the VARYING Phrase of a PERFORM Statement Having More Than Two Conditions 6–110
6–4. Flowchart for a Format 1 SEARCH Statement Containing Two WHEN Phrases 6–129

9–1. Compilation File 9–1

Tables

2–1. COBOL Character Set 2–1
2–2. Examples of MOVE ALL Literal 2–10
2–3. Examples of Nonnumeric Literals 2–12

4–1. Valid Combinations of Status Keys 1 and 2 for Sequential Files 4–24
4–2. Valid Combinations of Status Keys 1 and 2 for Relative Files 4–26
4–3. Valid Combinations of Status Keys 1 and 2 for Indexed Files 4–28

5–1. Classes and Categories of Data Items 5–6
5–2. STANDARD Label Record Description 5–26
5–3. Data Categories and Editing Methods Allowed 5–54
5–4. Data Item Values and Results Produced by Editing Sign Control Symbols 5–55
5–5. Editing Application of the PICTURE Clause 5–57

6–1. Combination of Symbols in Arithmetic Expressions 6–12
6–2. Truth Table for Logical Operators 6–21
6–3. Combinations of Conditions, Logical Operators, and Parentheses 6–22
6–4. Relationship of Files and the Formats of the CLOSE Statement 6–63
6–5. Validity of Moves between Various Categories of Data Items 6–94
6–6. Permissible Statements 6–99
6–7. Valid Operand Combinations for the SET...TO Statement 6–137

7–1. File Attributes 7–5
7–2. KIND Attribute Values 7–14
7–3. Port and Subport Attribute Summary 7–22

8–1. Minimum Key Buffer Sizes 8–17
8–2. Indexed File Status Conditions 8–20
8–3. Indexed File Fatal Error Messages 8–21
8–4. Indexed File Shared Error Messages 8–22

9–1. Valid INSERT Values for Cardless Compiles 9–4
9–2. Meanings of INSERT Values for Cardless Compiles 9–4
9–3. Compiler Files and Formats 9–5
9–4. Meaning of Federal Level Values 9–13
9–5. Federal Levels 9–14
9–6. COBOL Compiler Limits 9–24

C–1. USASCII X3.4–1963 ..... C–2
C–2. USASCII X3.4–1967 ..... C–3
C–3. Extended Binary Coded Decimal Interchange Codes (EBCDIC) C–4
C–4. BCL Language Codes C–8

Tables

D–1. Reserved Memory Layout D–1

F–1. OBJECT-COMPUTER Name Conversions F–9
F–2. Hardware Name Changes F–11
F–3. Invalid Communication Devices F–12
F–4. NN Option Conversions F–13
F–5. RECORDING MODE Conversions - STANDARD F–16
F–6. RECORDING MODE Conversions - NON-STANDARD F–16
F–7. LABEL RECORD Conversions F–17
F–8. Abbreviation Conversions F–19
F–9. CLOSE Conversions for Mass Storage Files F–24
F–10. CLOSE Conversions for Non-Mass Storage Files F–25
F–11. CLOSE REEL Conversions for Non-Mass Storage Files F–25
F–12. TRACE Statement Conversions F–35
F–13. ANSI Reserved Words F–37
F–14. ANSI Report Writer Reserved Words F–38
F–15. Unisys Reserved Words F–39

Examples

1–1.	Keywords	1–3
1–2.	Brackets and Braces	1–4
1–3.	Ellipsis	1–5
1–4.	Comment Lines	1–8
1–5.	Continuation Lines	1–9
1–6.	Debugging Lines	1–10
1–7.	Coding for Readability	1–12
2–1.	Key Words	2–7
3–1.	Coding the Identification Division	3–2
3–2.	Date Compiled Paragraph	3–3
4–1.	Coding the Environment Division	4–33
5–1.	Level Number Construction for Record in Figure 5–2	5–5
5–2.	One-Dimensional Table	5–9
5–3.	Three-Dimensional Table	5–10
5–4.	Table Initialization	5–11
5–5.	Coding FD and DATA RECORDS	5–24
5–6.	Coding the VALUE OF Clause	5–32
5–7.	Elementary and Group Items	5–36
5–8.	Coding Condition Names	5–40
5–9.	Legal Array Definition	5–61
5–10.	Legal Array Definition	5–61
5–11.	Illegal Array Definition	5–61
5–12.	Examples of RENAMES	5–63
5–13.	Coding the Working-Storage Section	5–72
E–1.	Use of STRING and UNSTRING	E–1
E–2.	USE Procedure to Handle a Parity Error	E–4
E–3.	Programmatic Handling of an Indexed File	E–5
E–4.	Binding Example—Creating ICMs	E–8
E–5.	Binding Example—Calling ICMs	E–9
E–6.	Output Produced by Dollar Option Map	E–10
E–7.	USE Procedure to Handle Tape Labels	E–16

Section 1
Overview and Format

This section provides an introduction to the format of the V Series COBOL ANSI-74 language.

Overview

The long list of advantages of COBOL is derived chiefly from its ability to permit the programmer to state the problem solution in English. The programming language reads much like ordinary English prose and can provide automatic program and system documentation. When users adopt in-house standards for elements within files and choose data names carefully before attempting to program a system, they obtain maximum documentational advantage of the language.

To the computer user, Unisys V Series COBOL offers the following major advantages:

- Fast and efficient program implementation
- Accelerated programmer training and simplified retraining requirements
- Reduced conversion costs when changing to a Unisys computer
- Significant ease of program modification
- Standardized documentation
- Easy-to-use documentation
- Efficient object program code
- Segmentation capability that sets a high maximum allowable program size
- A high degree of sophistication in program design because the language includes debugging capabilities
- A comprehensive source program diagnostic capability

Source Program

A program written in COBOL, called a source program, is accepted as input by the COBOL compiler. The compiler first verifies that the program satisfies all rules outlined in this manual. Then it translates the source program language into an object program capable of communicating with the computer and directing it to process the desired data. If source corrections become necessary, you can make appropriate changes and recompile the program. Thus, the source file always reflects the object program being operationally compiled.

Structure

A COBOL source program is divided into four parts, called divisions, in the following order:

- Identification
- Environment
- Data
- Procedure

The purpose of the Identification Division is to identify the program and to provide an overall description of the program.

The Environment Division consists of two sections. The Configuration Section specifies the equipment being used. The Input-Output Section associates files with the hardware devices that are used for their operation. This section also furnishes the compiler with information about mass storage parameters.

The Data Division is used to describe data elements that the object program is to manipulate or create. These data elements may be items within files, records or program work areas, and constants.

The Procedure Division defines the necessary steps to accomplish the desired task using the data defined in the Data Division.

Compiler Problems

If you encounter any problems when using the COBOL compiler or when executing the resulting object code, be sure to contact your local Unisys representative. Submit documentation to Unisys to enable the company to recreate the problem. The COBOL source for the failing program must be submitted to Unisys in machine-readable format, such as magnetic tape. Be sure to include all necessary library files.

For DMSII programs, also submit the DDF (data definition file) and DASDL source files.

Format

This manual includes Unisys extensions to the COBOL Standard. In Volume 1, Sections 1 through 6, Appendix A, and the Glossary, blue print marks those portions of the COBOL language that are designated as Unisys extensions. Some portions of the manual deal exclusively with Unisys extensions or areas of the COBOL standard that are implementor-defined. These sections are printed in black to improve readability. The sections that describe Unisys extensions include Sections 7 through 10 and Appendices B through F of Volume 1 and all of Volume 2: *Product Interfaces* (except Appendix A and the glossary).

Notation Used in Formats

The paragraphs that follow enable the reader to interpret the COBOL syntax as presented in this manual.

A general format is the specific arrangement of a clause or statement. A clause or statement consists of elements such as uppercase words, level numbers, brackets, braces, connectives, and special characters. Throughout this document, general formats are shown adjacent to information defining the clauses or statements.

When more than one specific arrangement is permitted, the general format is separated into numbered variations. Unless explicitly stated otherwise, clauses must be written in the sequence given in the general formats. (Optional clauses, when used, must appear in the sequence shown.)

Words

All underlined uppercase words are called *keywords* and are required when the functions that they are a part of are used. Uppercase words that are not underlined are optional to the user and may or may not be present in the source program. Uppercase words, whether underlined or not, must be spelled correctly.

Lowercase words are generic terms used to represent COBOL words, literals, picture character-strings, comment-strings, or a complete syntactical entry that the user supplies. Where generic terms are repeated in a general format, a number or letter appended to the term identifies that term for explanation.

Consider the following example:

```
RECORD CONTAINS integer-1 TO integer-2
CHARACTERS DEPENDING ON data-name-1
```

Example 1–1. Keywords

The words RECORD, TO, DEPENDING, and ON are keywords that must be included. The words CONTAINS and CHARACTERS are optional words so, they may be omitted. *Integer-1*, *Integer-2*, and *data-name-1* are generic terms meaning that the user must supply a user-defined word.

Level Numbers

When specific level numbers appear in data description entry formats, those specific level numbers are required when such entries are used in a COBOL program. In this document, the form *01, 02, ..., 09* is used to indicate level numbers 1 through 9.

Brackets and Braces

When a portion of a general format is enclosed in brackets ([ ]) the user may either include or omit that portion. When a portion of a general format is enclosed in braces ({ }), one of the options contained within the braces must be selected. In both cases, a choice is indicated by vertical stacking of the possibilities.

When brackets or braces enclose a portion of a format but only one possibility is shown, the brackets or braces delimit that portion of the format to which a following ellipsis applies. If an option within braces does not contain keywords, then the option is a default option.

Consider the following example:

```
BLOCK CONTAINS [integer-1 TO] integer-2 { RECORDS    }
                                        { CHARACTERS }
```

Example 1–2. Brackets and Braces

The brackets indicate that the entry *integer-1 TO* is optional, so it may be omitted. The braces indicate that the user must choose between RECORDS and CHARACTERS. Note, however, that CHARACTERS is an optional word. Therefore, the default option CHARACTERS is assumed implicitly unless RECORDS is explicitly stated. That is, the statement *BLOCK 5* is correct and equivalent to *BLOCK CONTAINS 5 CHARACTERS.*

Ellipsis

In text, the ellipsis (...) shows that a portion of a source program is omitted. This meaning becomes apparent in context.

In the general formats, the ellipsis indicates where repetition may occur. The repeatable portion of the format is determined as follows:

> Given an ellipsis (...) in a clause or statement format, find the right bracket (]) or brace (}) immediately to the left of the ellipsis. Then find the logically matching left bracket ([) or brace ({). The ellipsis applies to the words between these delimiters.

Consider the following example:

```
[ ; SAME [RECORD] AREA FOR file-name-3 {,file-name-4}... ]...
```

Example 1–3. Ellipsis

The first ellipsis indicates that you may repeat the phrase *file-name-4*. The second ellipsis means that the entire clause from SAME through the phrase *file-name-4* may be repeated.

Punctuation

The comma and semicolon are used as separators to improve the readability of the program. Suggested uses are shown in formats; however, their use is optional. In the source program, comma, semicolon, and space separators are interchangeable.

If desired, a semicolon or comma may be used between statements in the Procedure Division.

A separator period must terminate paragraphs within the Identification and Procedure Divisions and entries within the Environment and Data Divisions.

In the preceding example, the word *SAME* and the phrase *file-name-4* may be preceded by a semicolon, a comma, or space.

Special Characters

The characters +, −, >, <, =, are required when they appear in formats.

Coding Form

The reference format provides a standard method for describing COBOL source programs. It is described in terms of positions for each input-output medium. Within these definitions, each COBOL compiler accepts source programs written in reference format and produces an output listing of the source program input in reference format. The following discussion outlines this reference format.

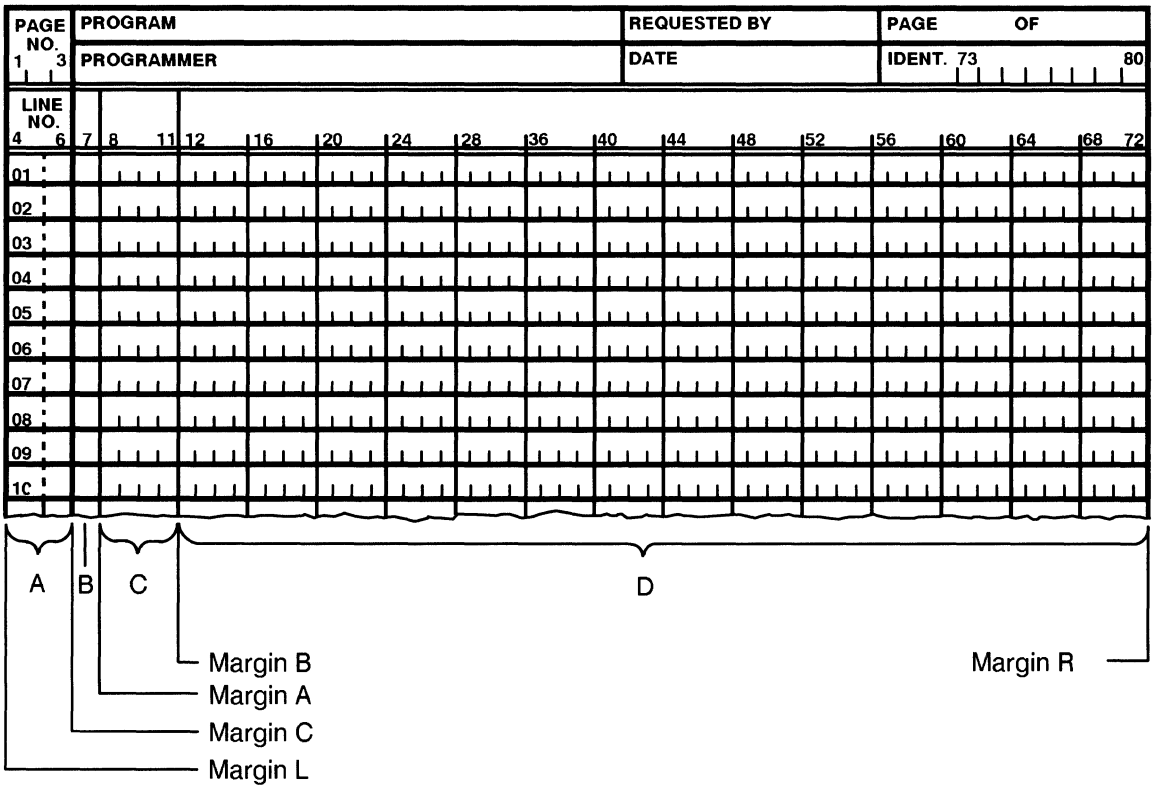
The divisions of a source program must be ordered as follows: the Identification Division, the Environment Division, the Data Division, and the Procedure Division. Each division must be written according to the rules that follow.

The rules for spacing given in the following discussion of the reference format take precedence over all other spacing rules.

Field Definitions

The same coding form is used for all four divisions of the COBOL program. Figure 1–1 and the paragraphs that follow describe the various fields of this coding form.

COBOL CODING FORM



A. 1 - 6 Sequence field. Used for a sequence number

B. 7 Indicator area. Used to denote a continuation line (-), a comment line (*or /), a debugging line (D), or a dollar option ($).

C. 8 - 11 Area A. Items which must begin in this area are: division headers; section headers; paragraph headers; the key words DECLARATIVES and END DECLARATIVES; level indicators FD and SD; and the level numbers 01 and 77.

D. 12 - 72 Area B. All other items must begin and end at some position in this area.

Figure 1–1. COBOL Coding Form and Reference Format

1–6 Sequence Area

A six-digit sequence number may be used to label a source program line.

7 Indicator Area

An entry in this area denotes that this line is either a comment, a continuation of the previous line, a line used only for debugging, or a dollar option.

Comment Line

A comment line is any line with an asterisk (*) in the indicator area of the line. A comment line can appear as any line in a source program after the Identification Division header. Any combination of characters from the computer's character set may be included in area A and area B of that line. The asterisk and the characters in area A and area B appear on the listing but only serve as documentation.

Successive comment lines are allowed. Continuation of comment lines is permitted, but each line must contain an asterisk (*) in the indicator area.

A special form of comment line represented by a stroke (/) in the indicator area of the line causes page ejection prior to printing the comment.

Example 1–4 shows the use of comment lines.

```
*
*THIS ENTRY ALLOWS DOCUMENTATIONAL DATA
*    OR BLANK LINES TO BE
*INSERTED AT ANY POINT IN A SOURCE PROGRAM


/THIS ENTRY WILL BE PRINTED AT THE TOP OF A NEW PAGE
```

Example 1–4. Comment Lines

Continuation Line

A continuation line is any line with a hyphen (-) in the indicator area of the line. Whenever a sentence, entry, phrase, or clause requires more than one line, you may continue it by placing a hyphen in column 7 of the subsequent line(s) and placing the continued information in area B, columns 12–72, of that line.

A hyphen in the indicator area of a line indicates that the first nonblank character in area B of the current line follows the last nonblank character of the preceding line without any intervening space. If the continued line contains a nonnumeric literal without a closing quotation mark, the first nonblank character in area B on the continuation line must be a quotation mark ("). The continuation then starts with the character immediately after that quotation mark.

If the continued line contains a hexadecimal literal, the first nonblank character in area B on the continuation line must be a commercial at sign (@). The continuation then starts with the character immediately after that commercial at sign. All spaces at the end of the continued line are part of the literal, except for those that are hexadecimal. Area A (columns 8 through 11) of a continuation line must be blank.

If there is no hyphen in the indicator area of a line, you can assume that the last character in the preceding line is followed by a space.

Example 1–5 shows the use of continuation lines.

```
      SELECT MASTERFILE  ASSIGN TO DISKPACK OR
-     ORGANIZATION IS SEQUENTIAL.

 01   WARNING-MESSAGE PIC X(24)  VALUE IS "WRONG ENTRY FOR THIS KEY
-     "".

 77   HEADER-LINE     PIC X(60)  VALUE IS "    JANUARY    FEBRUARY
-     "   MARCH    APRIL    MAY    JUNE    ".

 77   HEX-LITERAL     PIC X(30)  VALUE IS @AAAAAAAAAABBBBBBBBBBCCCC
-     @CCCCCC@.
```

Example 1–5. Continuation Lines

Debugging Line

A debugging line is any line with a D in its indicator area. Any debugging line that consists solely of spaces from column 8 through column 72 is considered a blank line. A debugging line is permitted in the program only after the OBJECT-COMPUTER paragraph.

A debugging line has all the characteristics of a comment line if the WITH DEBUGGING MODE clause is not specified in the SOURCE-COMPUTER paragraph. If this clause is specified in the SOURCE-COMPUTER paragraph, debugging lines are compiled as part of the program. Therefore, the contents of a debugging line must allow a syntactically correct program to be formed whether or not the debugging lines are considered comment lines.

Successive debugging lines are allowed. Each continuation line must contain a D in the indicator area. Also, character strings may not be broken across two lines.

Example 1–6 illustrates some debugging lines.

```
 SOURCE-COMPUTER   V380 WITH DEBUGGING MODE.
 .
 .
 .
 WORKING-STORAGE SECTION.
D77  PERFORMANCE-COUNT   PIC 9(4)
D77  BAD-RECORDS         PIC 9(4)
D77  RATIO               PIC 9(4).99.
 .
 .
 .
 PROCEDURE DIVISION.
 A.  OPEN INPUT INPUT-FILE.
D    MOVE ZEROS TO PERFORMANCE-COUNT, BAD-RECORDS, RATIO.
 B.  READ INPUT-FILE  AT END GO TO C.
D    ADD 1 TO PERFORMANCE-COUNT.
D    If IN-KEY NOT NUMERIC ADD 1 TO BAD-RECORDS.
 .
 .
 .
     GO TO B.
 C.  CLOSE INPUT-FILE DISMISS.
D    DIVIDE PERFORMANCE-COUNT BY BAD-RECORDS GIVING RATIO.
 .
 .
 .
```

Example 1–6. Debugging Lines

Compiler Control Options

A compiler control option, also called a dollar option, is any line with a dollar sign ($) in the indicator area of the line. This line is used to identify for the compiler the available compiler control options that are required during compilation.

Refer to Section 9, "Compiler Operation," for a complete discussion of the compiler control options and their use.

8–11 Area A

Division, section, and paragraph headers must begin in area A. A division header consists of the division name (IDENTIFICATION DIVISION, ENVIRONMENT DIVISION, DATA DIVISION, or PROCEDURE DIVISION) followed by a period. A section header consists of the section name followed by a space and then the word *SECTION* followed by a period. A paragraph header consists of a paragraph name followed by a period.

Within the Data Division, the level indicators (FD,SD) and the level numbers 01 and 77 must also begin in area A, followed by an associated name and appropriate descriptive information.

The keyword DECLARATIVES and the keywords END DECLARATIVES that precede and follow the declaratives portion of the Procedure Division must appear on lines by themselves in area A and be followed by a period and a space.

12–72 Area B

The first sentence or entry in a paragraph begins either on the same line as the paragraph header or paragraph name or in area B of the next nonblank line that is not a comment line. Successive sentences or entries begin either in area B of the same line as the preceding sentence or entry or in area B of the next nonblank line that is not a comment line.

In the Data Division, all level numbers other than 01 and 77 may begin anywhere in either area A or area B.

73–80 Identification

This area is treated as a comment only. You may enter here any characters valid on the system. For example, use this area to identify the program or particular source lines.

Blank Lines

A blank line is one that contains no entries in the indicator area, in area A, or in area B. A blank line may appear anywhere in the source program except immediately preceding a continuation line.

Sample Coding Form Usage

Level numbers 01 and 77 must be coded in area A. For readability, Unisys suggests that you code other level numbers in area B. If you indent each successively higher level number four positions, it makes the coding easier to follow, and structure is readily apparent. Using odd numbered level numbers permits easy patching of record descriptions.

Coding repetitive information in the same columns makes the program easier to read; for example, PIC in columns 36 through 38, VALUE in columns 52 through 56.

Example 1–7 shows an example of coding for readability.

```
*
* INDENTATION IMPROVES PROGRAM READABILITY
*
 01  INPUT-RECORD.
     03  AMOUNT              PIC 9(5)V99.
     03  AMOUNT-OUT          PIC 9(5)V99.
     03  FACTORS             PIC 9(3)V9(5).
     03  PERCENT             PIC V999.
     03  NAME-CITY           PIC X(10).
     03  CODR                PIC XX.
     03  DATER.
         05  MONTH           PIC 99.
         05  DAY             PIC 99.
         05  YEAR            PIC 99.
             88  CUR-DECADE                  VALUE
                 60 THRU 69.
      03  FILLER             PIC X(33).
 66  IN-DATE  RENAMES  MONTH THRU YEAR.

*                            NOTE
*          THE ABOVE 88 LEVEL IS CONTINUED ON THE FOLLOWING
*          LINE; HOWEVER, A DASH MUST NOT APPEAR IN COLUMN 7,
*          AS THE COMPILER CONTINUES TO SCAN THE FOLLOWING
*          LINE IN AN EFFORT TO SATISFY THE VALUE REQUIREMENT.
```

Example 1–7. Coding for Readability

Section 2
Language Elements

COBOL is a language based on English. It is composed of words, statements, sentences, and paragraphs. This section presents the rules to be followed in using this language. The remainder of this manual explains how to form different constructs with the elements defined in this section.

Character Set

The basic unit of the language is the character. The set of characters used to form COBOL character strings and separators includes the letters of the alphabet, digits, and special characters as shown in Table 2–1. The character set consists of 52 characters. In the case of nonnumeric literals, comment entries, comment lines, and other text enclosed in quotation marks, the character set includes the computer's entire character set.

The character set required is shown in Table 2–1.

Table 2–1. COBOL Character Set

Character	Meaning
0 through 9	Digit
A through Z	Letter
Space	(blank)
+	Plus sign
–	Minus sign (hyphen)
*	Asterisk
/	Stroke (virgule, slash)
=	Equal sign
$	Currency sign
,	Comma (decimal point)
;	Semicolon
.	Period (decimal point)
"	Quotation mark

continued

Table 2–1. COBOL Character Set (cont.)

Character	Meaning
(	Left parenthesis
)	Right parenthesis
>	Greater-than symbol
<	Less-than symbol
@	Commercial at

Language Structure

The characters of the language are linked together to form character strings and separators. A separator is a string of one or more punctuation characters. A separator can be linked with another separator or with a character string. However, a character string can be linked only with a separator. Linking together character strings and separators forms the text of a source program.

Separators

The rules for formation of punctuation characters and spaces as separators are as follows:

- Space.

  More than one space may be used as separators.

- Comma, semicolon and period.

  Only one period may be used as a separator. Of these three, only the comma is followed by a space when used immediately after a numeric literal or PICTURE string. In such a case, the comma must be followed by a space to clarify its role as a separator. This rule holds even if the phrase DECIMAL POINT IS COMMA is used.

- Right and left parenthesis

  Parentheses may appear only in balanced pairs of left and right parentheses to delimit subscripts, indices, arithmetic expressions, or conditions.

- Quotation mark.

  Quotation marks may appear only in balanced pairs to delimit nonnumeric literals except when the literal is continued. An opening quotation mark must be immediately preceded by one of these separators: space, comma, semicolon, or left parenthesis. A closing quotation mark must be immediately followed by one of these separators: space, comma, semicolon, period, or right parenthesis.

- Pseudotext delimiters (two contiguous equal sign characters [==]).

  Pseudotext delimiters may appear only in balanced pairs to delimit pseudotext. An opening pseudotext delimiter must be immediately preceded by a space; a closing pseudotext delimiter must be immediately followed by one of these separators: space, comma, semicolon, or period.

- Commercial at sign (@).

  The @ character may appear only in balanced pairs to delimit hexadecimal literals. An opening @ character must be preceded immediately by one of these separators: space, comma, semicolon, or left parenthesis. A closing @ character must be immediately followed by one of these separators: space, comma, semicolon, period, or right parenthesis.

- Space

  This separator may immediately precede all separators except the following:

  - Closing quotation mark. In this case, a preceding space is part of the nonnumeric literal and not a separator.
  - Opening pseudotext delimiter, where the preceding space is required.
  - Those specified by reference format rules (see "Coding Form" in Section 1).

  The separator space may immediately follow any separator except the opening quotation mark. In this case, a following space is considered part of the nonnumeric literal and not a separator.

Any punctuation character that is part of the specification of a PICTURE character string or numeric literal is considered a symbol used in the specification of that PICTURE character string or numeric literal, not as a punctuation character. PICTURE character strings are delimited only by the separators space, comma, semicolon, or period.

The rules established for the formation of separators do not apply to nonnumeric literals, comment entries, or comment lines.

Character Strings

A character string is a character or sequence of contiguous characters that form a COBOL word, a literal, a PICTURE character string, or a comment entry. A character string is delimited by separators.

COBOL Words

A COBOL word is a character string of not more than 30 characters, which forms a user-defined word, a system name, or a reserved word. A given COBOL word can appear in a source program either as both a user-defined word and a system name or as both a system name and reserved word. However, a given COBOL word cannot be both a user-defined word and a reserved word.

User-Defined Words

A user-defined word is a COBOL word that you must supply to satisfy the format of a clause or statement. Each character of a user-defined word is selected from the set of characters A through Z, 0 through 9, and the hyphen (-). However, the hyphen cannot appear as the first or last character of a word.

The following are types of user-defined words:

Alphabet name
Condition name
Data name
Family name
File name
Index name
Level numbers
Mnemonic name
Paragraph name
Program name
Record name
Routine name
Section name
Segment numbers
Text name

With the exception of paragraph names, section names, level numbers, and segment numbers, all user-defined words must contain at least one alphabetic character.

Grouping these words indicates sets. All user-defined words except segment numbers and level numbers can belong to only one of these disjoint sets. Furthermore, all user-defined words within a given disjoint set except for segment numbers and level numbers must be unique. Uniqueness occurs when no other user-defined word in the same source program has identical spelling or punctuation or when you qualify the word. (See "Data Qualification" in Section 5). Segment numbers and level numbers need not be unique.

Alphabet Name
An alphabet name is a user-defined word in the SPECIAL-NAMES paragraph of the Environment Division that names a specific character set and/or collating sequence.

Condition Name

A condition name is a user-defined word that you assign to a specific value, set of values, or range of values within the complete set of values that a data item may assume. The data item itself is called a conditional variable.

Use a condition name only in conditions as an abbreviation for the relation condition; this relation condition states that the associated conditional variable is equal to one of the set of values to which that condition name is assigned.

You can define condition names in the Data Division or in the SPECIAL-NAMES paragraph within the Environment Division, where you must assign a condition name to the ON STATUS or OFF STATUS, or both, of implementer-defined switches.

Switches are defined in "Special-Names Paragraph" in Section 4, "Environment Division".

Data Name
A data name is a user-defined word that names a data item described in a data description entry in the Data Division. When used in the general formats, a data name represents a word which can neither be subscripted, indexed, nor qualified unless specifically permitted by the rules for that format.

Family name

A family name is a user-defined word that names a pack on which a COBOL library file resides.

File name

A file name is a user-defined word that names a file connector described in a file description entry or a sort-merge file description entry.

Index name

An index name is a user-defined word that names an index associated with a specific table.

Level number

A level number is a user-defined word that indicates the position of a data item in the hierarchical structure of a logical record, or indicates special properties of a data description entry. A level number is expressed as a 1- or 2-digit number. Level numbers in the range 1 through 49 indicate the position of the data item in the hierarchical structure of a logical record. Level numbers in the range 1 through 9 can be written either as a single digit or as a zero followed by a significant digit. Level numbers 66, 77, and 88 identify special properties of a data description entry.

Mnemonic Name

A mnemonic name is a user-defined word that assigns a user-defined word to an implementer name. These associations are established in the SPECIAL-NAMES paragraph of the Environment Division. An implementer name is a system name that refers to a particular feature available on the Unisys V Series COBOL compiler.

Paragraph Name

A paragraph name is a user-defined word that names a paragraph in the Procedure Division. Paragraph names are equivalent if they are composed of the same sequence of the same number of digits and/or characters.

Program name

A program name is a user-defined word that identifies a COBOL source program.

Record name

A record name is a user-defined word that names a record described in a record description entry in the Data Division.

Routine name

A routine name is a user-defined word that identifies a procedure written in a language other than COBOL.

Section Name

A section name is a user-defined word that names a section in the Procedure Division. Section names are equivalent if, and only if, they are composed of the same sequence of the same number of digits and/or characters.

Segment number

A segment number is a user-defined word that classifies sections in the Procedure Division for purposes of segmentation. Segment numbers can contain only the characters 0 through 9. A segment number can be expressed as a 1-, 2-, 3-, or 4-digit number.

Text name

A text name is a user-defined word that identifies library text.

System Names

A system name is a COBOL word that is used to communicate with the operating environment. Each character used in a system name must be from the sets of characters A through Z, 0 through 9, and the hyphen (-). The hyphen cannot be the first or last character.

There are two types of system names:

- Computer name
- Implementer name

A system name can belong to only one of these two types within a given implementation.

Computer name

A system name that identifies the computer upon which the program is to be compiled or run.

Implementer name

A system name that refers to a particular feature available on that implementer's computing system.

Reserved Words

A reserved word is a COBOL word that is one of a specified list of words that can be used in COBOL source programs, but that cannot appear in the programs as user-defined words. This rule applies to the entire COBOL source program.

There are six types of reserved words:

- Keywords
- Optional words
- Connectives
- Special registers
- Figurative constants
- Special-character words

Keywords

A keyword is a word that is required when its format is used in a source program. Within each format, such words are uppercase and underlined.

Keywords are of three types:

- COBOL Verbs such as ADD and READ
- Required words that appear in statement and entry formats
- Words that have a specific functional meaning such as NEGATIVE and SECTION

Note the keywords in Example 2–1:

```
RECORD CONTAINS [integer-1 TO] integer-2
       CHARACTERS [DEPENDING ON data-name-1]
```

Example 2–1. Key Words

RECORD is a keyword, TO is a keyword when the integer-1 phrase is used, and DEPENDING ON are keywords when the DEPENDING clause is used.

Optional Words

Within each format, uppercase words that are not underlined are optional words. The presence or absence of an optional word does not alter the semantics of the COBOL program in which it appears.

In the preceding example, CONTAINS and CHARACTERS are optional words.

Connectives

There are three types of connectives:

- Qualifier connectives, used to associate a data name, a condition name, a text name, or a paragraph name with its qualifier: OF, IN.
- Series connectives that link two or more consecutive operands: separator comma (,) or separator semicolon (;).
- Logical connectives used in the formation of conditions: AND, OR.

Special Registers

Some reserved words refer to special registers. Special registers are certain compiler-generated storage areas primarily used to store information produced in conjunction with the use of specific COBOL features.

The following are the special registers:

- DATE
- DAY
- DEBUG-ITEM
- TIME
- TIMER
- TODAYS-DATE
- TODAYS-NAME

The DEBUG-ITEM is described in "USE statement" in Section 6. The other items are described in "ACCEPT Statement" in that section.

Figurative Constants

Figurative constants are certain reserved words used in referring to specific constant values. These words must not be bounded by quotation marks. The singular and plural forms of figurative constants are equivalent and may be used interchangeably.

The figurative constant values and the reserved words used to reference them are as follows:

ZERO ZEROS ZEROES	Represent the value *0* or one or more instances of the character *0*, depending on context.
SPACE SPACES	Represent one or more instances of the *space* character from the computer's character set.
HIGH-VALUE HIGH-VALUES	Represent one or more instances of the character that has the highest ordinal position in the collating sequence of the program. In EBCDIC, this value is all bits on, hexadecimal FF.
LOW-VALUE LOW-VALUES	Represent one or more instances of the character that has the lowest ordinal position in the collating sequence of the program. In EBCDIC, this value is all bits off, hexadecimal 00.
QUOTE QUOTES	Represent one or more quotation mark characters ("). You cannot use the word QUOTE or QUOTES in place of a quotation mark in a source program to bound a nonnumeric literal. Thus, QUOTE ABD QUOTE is not the same as the nonnumeric literal *"ABD"*.
ALL literal	Represents one or more of the string of characters comprising the literal. The literal must be - a nonnumeric - a hexadecimal - a figurative constant other than ALL literal. When a figurative constant is used, the word ALL is redundant. Use it for readability only. Table 2–2 shows examples of this ALL construct used in a MOVE ALL literal construct.

Table 2–2. Examples of MOVE ALL Literal

ALL Literal	Size of Receiving Field (Characters)	Resultant Value of Receiving Field
ALL "ABC"	7	ABCABCA
ALL "3"	5	33333
ALL "HI-LO"	12	HI-LOHI-LOHI
ALL QUOTES	3	" " "
ALL "LIMIT"	4	LIMI
ALL SPACES	5	

Note: *Because QUOTES and SPACES are figurative constants, MOVE ALL QUOTES and MOVE QUOTES yield the same results, as do MOVE ALL SPACES and MOVE SPACES.*

When a figurative constant represents a string of one or more characters, the compiler determines the length of the string from context, according to the following rules:

- When a figurative constant is associated with another data item, as when the figurative constant is moved to or compared with another data item, the string of characters specified by the figurative constant is repeated character by character on the right until the resultant string equals in characters the size of the associated data item. The compiler does so independently before applying any JUSTIFIED clause that may be associated with the data item.
- When a figurative constant is not associated with another data item, as when the figurative constant appears in a DISPLAY, STRING, STOP, or UNSTRING statement, the length of the string is one character.

You may use a figurative constant wherever a literal appears in a format, with one exception. Whenever the literal is restricted to numeric characters, the only figurative constant permitted is ZERO (ZEROS, ZEROES).

The program collating sequence determines the association of an actual character with the figurative constants HIGH-VALUE(S) or LOW-VALUE(S). The sequence is specified when these constants are used in the source program. (See "OBJECT-COMPUTER Paragraph" and "SPECIAL-NAMES Paragraph" in Section 4).

Each reserved word used to reference a figurative constant value is a distinct character string with the exception of the construct "ALL literal". This construct is composed of two distinct character strings.

Special Character Words

Special character words are reserved words that are arithmetic operators or relation characters. The following are special character words:

- + (Plus sign)
- − (Minus sign)
- * (Asterisk)
- / (Slash (virgule, stroke))
- = (Equal sign)
- $ (Currency sign)
- , (Comma)
- ; (Semicolon)
- . (Period (decimal point))
- " (Quotation mark)
- ((Left parenthesis)
-) (Right parenthesis)
- > (Greater than symbol)
- < (Less than symbol)
- @ (Commercial at sign)

Literals

A literal is a character string whose value is implied by an ordered set of characters of which the literal is composed or by a reserved word that references a figurative constant. Every literal belongs to one of three types:

- numeric
- nonnumeric
- hexadecimal

Nonnumeric Literals

A nonnumeric literal is a character string delimited by quotation marks and consisting of any allowable character in the computer's character set. To represent one quotation mark character within a nonnumeric literal, you must use two contiguous quotation marks. The value of a nonnumeric literal in the object program is the string of characters itself, excluding the delimiting quotation marks. In addition, each embedded pair of contiguous quotation marks represents one quotation mark character. All other punctuation characters are part of the value of the nonnumeric literal; they are not separators.

Nonnumeric literals are in the alphanumeric category. Therefore, you must not enclose in quotation marks any literals used for arithmetic computation. The literal "−*7.7*" is nonnumeric. It is stored differently than the numeric literal −*7.7* (not enclosed in quotation marks).

Examples of nonnumeric literals are shown in Table 2–3.

Table 2–3. Examples of Nonnumeric Literals

Symbolic Source Program Nonnumeric Literal	Literal Stored by Compiler
"ANNUAL DUES"	ANNUAL DUES
"(MILES/GALLON)"	(MILES/GALLON)
"−123.456"	−123.456
"A""B"	A"B
""""LIMITATIONS""""	"LIMITATIONS"

Numeric Literals

A numeric literal is a character string consisting of the digits 0 through 9, the plus sign (+), the minus sign (−), and the decimal point (.).

Numeric literals may be from 1 through 160 digits in length when not used as arithmetic operands. The literal length of an arithmetic operand is from 1 through 98 digits. The rules for the formation of numeric literals are as follows:

- A literal must contain at least one digit.
- A literal must not contain more than one sign character.
  - If a sign is used, it must be the leftmost character of the literal.
  - If the literal is unsigned, the literal is positive.
- A literal must not have more than one decimal point. The decimal point is treated as an assumed decimal point and may appear anywhere within the literal except as the rightmost character.
  - If the literal contains no decimal point, the literal is an integer.
  - If a literal is enclosed in quotation marks, the compiler treats it as a nonnumeric literal.
- The value of a numeric literal is the algebraic quantity represented by the characters in the numeric literal. Every numeric literal is in the numeric category. The number of digits you specify is the size of a numeric literal in standard data format characters.

The following are examples of numeric literals:

```
12345
.005
+1.008
−0.0965
7842.1
```

Hexadecimal Literals

A hexadecimal literal is a character string delimited by commercial at signs (@) and consisting of characters selected from the hexadecimal digits 0 through 9 and A through F.

The category of a hexadecimal literal (4-bit numeric or 8-bit alphanumeric) is determined by the category of the data item with which it is associated in a COBOL statement. A hexadecimal literal is handled as if it were 4-bit numeric when the associated data item is numeric, whether its USAGE is COMPUTATIONAL or DISPLAY.

A hexadecimal literal is handled as if it were 4-bit numeric in the following conditions:

- In the VALUE clause, the associated data item is numeric.
- In the MOVE statement, the receiving data item is numeric or numeric edited.
- In the conditional expression of an IF, PERFORM, or SEARCH statement, the other relational operand is numeric.

A hexadecimal literal is handled as if it were an 8-bit alphanumeric when the associated data item is nonnumeric. Each character is represented by two hexadecimal digits. This representation requires an even number of digits in the hexadecimal literal. A hexadecimal literal is handled as if it were alphanumeric in the following conditions:

- The associated data item is not numeric in the VALUE clause.
- The receiving data item is alphanumeric, alphabetic, or alphanumeric-edited in the MOVE statement.
- The other relational operand is not numeric in the conditional expression of an IF, PERFORM, or SEARCH statement.
- It appears in an INSPECT, STRING, UNSTRING, DISPLAY, STOP, DISABLE, or ENABLE statement.
- It appears in the ALL figurative constant.

A hexadecimal literal may also appear in a COPY statement. In this case, the hexadecimal literal is not associated with a type.

The following restrictions apply to hexadecimal literals:

- A hexadecimal literal is not allowed as an arithmetic operand either in an ADD, SUBTRACT, MULTIPLY, or DIVIDE statement or in an arithmetic expression in a COMPUTE statement or conditional expression.
- A hexadecimal literal is not allowed as a subscript or index.
- An identifier assigned a hexadecimal literal will usually not compare as either numeric or alphabetic in a class condition test.
- The length of the literal must be from 1 through 160 digits when a hexadecimal literal is handled as if its category were computational. The length of the literal must be from 1 through 290 digits when a hexadecimal literal is handled as if it were nonnumeric.

Numeric Data Items

Numeric data item sizes must be within a range of 1 through 99999 for uses other than arithmetic operations; arithmetic operands cannot be greater than 98 digits.

Alphanumeric Data Items

Alphanumeric data item sizes must be within a range of 1 through 499999

PICTURE Character Strings

A PICTURE character string consists of certain combinations of symbols in the COBOL character set. See "PICTURE Clause" in Section 5 for a discussion of the PICTURE character string and the rules that govern its use.

Any punctuation character that is part of the specification of a PICTURE character string is not considered as a punctuation character, but rather as a symbol used in the specification of that PICTURE character string.

Comment Entries

A comment entry is an entry in the Identification Division that may be any combination of characters from the computer's character set.

Section 3
Identification Division

The first part or division of the source program is the Identification Division. Its function is to provide documentational information about the program, including identifying the source program, the date the program was written, and the date the compilation was accomplished.

Structure

The following is the structure of this division:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  program-name.
[AUTHOR.  [comment-entry] ... ]
[INSTALLATION.  [comment-entry] ... ]
[DATE-WRITTEN.  [comment-entry] ... ]
[DATE-COMPILED.  [comment-entry] ... ]
[SECURITY.  [comment-entry] ... ]
```

Figure 3–1. Structure of the Identification Division

Observe the following rules when forming the Identification Division:

- Begin the Identification Division with the reserved words *IDENTIFICATION DIVISION*, followed by a period and a space.
- Use any combination of the characters from the computer's character set when making the comment-entry. Do not continue the comment-entry by using the hyphen in the indicator area. However, you may use more than one line for the comment-entry.

With the exception of the DATE-COMPILED paragraph, the entire division is simply copied from the input source program and listed as output.

Coding Example

Example 3–1 shows an example of Identification Division coding.

```
IDENTIFICATION DIVISION
PROGRAM-ID.  SAMPLE.
AUTHOR.  UNISYS CORPORATION.
INSTALLATION.  PASADENA DEVELOPMENT CENTER.
DATE-WRITTEN.  FEBRUARY 28, 1990.
DATE-COMPILED.
SECURITY.  COMPANY CONFIDENTIAL.
```

Example 3–1. Coding the Identification Division

PROGRAM-ID Paragraph

The PROGRAM-ID paragraph gives the name by which a program is identified.

The following is the general format:

```
PROGRAM-ID.  program-name.
```

The PROGRAM-ID paragraph must contain the name of the program and be present in every program.

The program-name identifies the source program and all listings for that program. The program-name must conform to the rules for forming a user-defined word (see "User-Defined Words" in Section 2).

If the source program creates ICM files, then use the program-name in any CALL MODULE statement to invoke sections within this program (see Section 10, "COBOL Binding").

DATE-COMPILED Paragraph

The DATE-COMPILED paragraph provides the compilation date in the Identification Division source program listing.

The following is the general format:

```
DATE-COMPILED.  [comment-entry]
```

The comment-entry may be any combination of characters from the computer's character set. Do not use a hyphen in the indicator area to continue this comment-entry; however, you can use more than one line for the comment-entry. Also, the date the program is compiled is listed in the DATE-COMPILED entry as follows:

```
DATE-COMPILED. <year month day hh:mm>
```

Example 3–1 provides an example of the DATE-COMPILED paragraph.

```
DATE-COMPILED. 1989 OCTOBER 17 17:04
```

Example 3–2. Date Compiled Paragraph

Section 4
Environment Division

The Environment Division is the second division of a COBOL source program. This division specifies a standard method of expressing those aspects of a data processing problem that are dependent upon the physical characteristics of a specific computer. The Environment Division allows you to specify the configuration of the compiling computer and the object computer. In addition, you can specify information relating to input/output control, special hardware characteristics, and control techniques.

Structure

Two sections make up the Environment Division: the Configuration Section and the Input-Output Section.

The Environment Division must begin with the reserved words *ENVIRONMENT DIVISION*, followed by a period and a space.

Figure 4–1 shows the structure of this division:

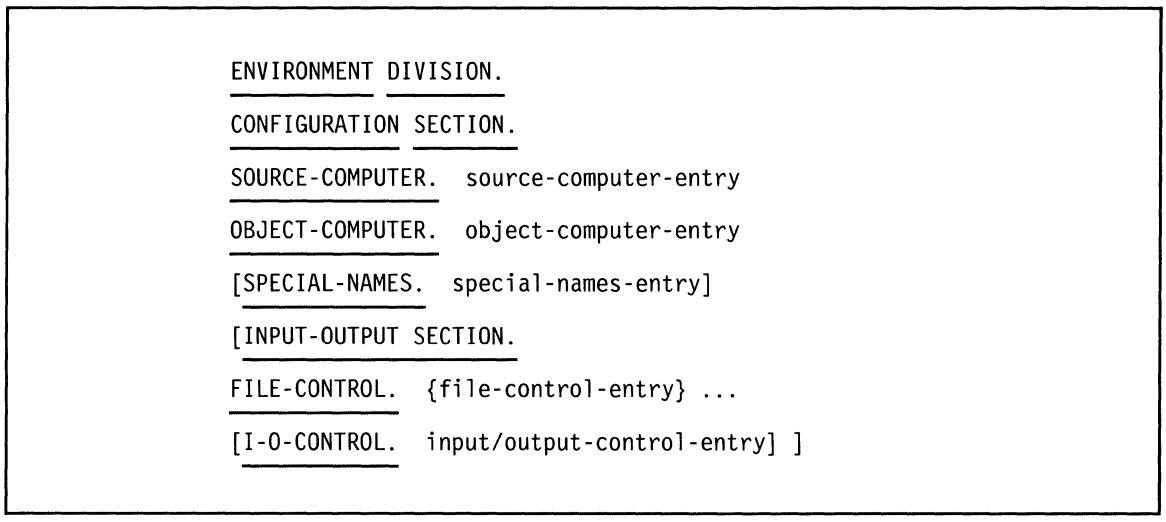


Figure 4–1. Structure of the Environment Division

Configuration Section

The Configuration Section deals with the characteristics of the source computer and the object computer. This section is divided into three paragraphs:

- The *Source-Computer* paragraph describes the computer configuration on which the source program is compiled.
- The *Object-Computer* paragraph describes the computer configuration on which the object program produced by the compiler is to be run.
- The *Special-Names* paragraph relates the implementer names used by the compiler to the mnemonic names used in the source program.

Source-Computer Paragraph

The Source-Computer paragraph identifies the computer upon which the program is to be compiled.

The following is the general format:

```
SOURCE-COMPUTER. computer-name [WITH DEBUGGING MODE].
```

Computer-Name Clause

The computer-name is a system name that is handled as a comment entry. It describes the computer upon which the source program is to be compiled.

WITH DEBUGGING MODE Clause

The WITH DEBUGGING MODE clause serves as a compile-time switch over the debugging lines, statements, and sections written in the program. The WITH DEBUGGING MODE clause indicates that all debugging lines (that is, lines with a D in column 7), statements, and sections are to be compiled. If this clause is not specified, debugging statements, lines, and sections are treated as comments at compile time.

SW6 is the run-time switch that governs whether the Use For Debugging section will be executed. If the WITH DEBUGGING MODE clause is not specified at compile time, switch SW6 is turned off and Use For Debugging statements, lines, and sections are not executed at run time. For further information on debugging, refer to Appendix D.

Object-Computer Paragraph

The Object-Computer paragraph identifies the computer on which the program is to be executed.

The following is the general format:

```
OBJECT-COMPUTER. computer-name [ , MEMORY SIZE integer { WORDS      } ]
                                                       { CHARACTERS }
                                                       { MODULES    }
    [ , STACK SIZE IS integer-2]
    [ , PROGRAM COLLATING SEQUENCE IS alphabet name]
    [ , SEGMENT-LIMIT IS segment-number]
```

The computer-name is a system name that identifies the hardware for which object code is to be generated.

The following are valid Object-Computer system names:

V-500
V-400
V-300
B-4900
B-3900
B-2900

MEMORY SIZE Clause

If the compiled program requires more than 150KB, the MEMORY SIZE clause is required and must specify a size greater than 150KB. Otherwise, the MEMORY SIZE clause is optional. It can be used as a guideline for determining the memory allocation required for running the object program.

The WORDS or MODULES option yields a minimum memory size of integer-times-2 characters.

STACK SIZE Clause

If you specify the STACK SIZE clause, integer-2 is two digits in size and indicates, in thousands, the number of digits desired for the program stack. If the size that

the compiler would otherwise assign is greater than integer-2, the program specification prevails.

PROGRAM COLLATING SEQUENCE Clause

If you specify the PROGRAM COLLATING SEQUENCE, this is the collating sequence associated with the alphabet name specified in that clause. It is used to determine the truth value of any nonnumeric comparisons explicitly specified in relation conditions and in condition name conditions. It is also applied to nonnumeric Sort or Merge keys unless you specify the COLLATING SEQUENCE phrase of the respective SORT or MERGE statement.

If the PROGRAM COLLATING SEQUENCE is not specified, the compiler uses the native collating sequence.

SEGMENT-LIMIT Clause

Ideally, all program segments having blank segment numbers or segment numbers ending in 0 through 049 should be configured as permanent segments. However, when insufficient memory is available to contain all permanent segments plus the largest overlayable segment, you must decrease the number of permanent segments. The SEGMENT-LIMIT feature provides you with a means to reduce the number of permanent segments in the program, while retaining the logical properties of fixed portion segments.

Segment number in the SEGMENT-LIMIT clause must be an integer ranging in value from 1 through 49.

When you specify the SEGMENT-LIMIT, only those segments having no segment numbers or having segment numbers ending in 0 up to, but not including, the segment number designated as the segment limit are considered permanent segments of the object program. These permanent segments are referred to as fixed, permanent segments. Segments ending in the designated segment limit up through 049 are fixed, overlayable segments. All others are independent segments.

When you omit the SEGMENT-LIMIT clause, all segments having blank segment numbers or segment numbers ending in 0 through 049 are considered permanent segments of the object program.

For further information regarding segmentation, refer to "Segmentation" in Section 6.

Special-Names Paragraph

The Special-Names paragraph provides a means of relating implementer names to user-specified names, and alphabet names to character sets as well as to collating sequences.

The following is the general format:

```
SPECIAL-NAMES.
[ , CHANNEL nn IS mnemonic name                                          ]
[ , ODT IS mnemonic name                                                 ]
[                                  { ON STATUS IS condition-name-1       }]
[                                  {   [ , OFF STATUS IS condition-name-2] }]...
[ , switch name                    {                                     }]
[       [ IS mnemonic name ]       { OFF STATUS IS condition-name-2       }]
[                                  {   [ , ON STATUS IS condition-name-1] }]

[                                  { STANDARD-1                          }]
[                                  { ASCII                               }]
[                                  { NATIVE                              }]
[                                  { EBCDIC                              }]
[ , alphabet name IS               {  [                                  }]
[                                  {  |literal-1  [ { THROUGH } literal-2 }]]
[                                  {  |           { THRU    }           }]
[                                  {  |           ALSO literal-3         }]
[                                  {  |           [ , ALSO literal-4 ...]...  ]...  }]...
[ , CURRENCY SIGN IS literal-9]
[ , DECIMAL-POINT IS COMMA].
```

Clauses in the Special-Names paragraph, though optional, must appear in the order shown.

CHANNEL Clause

Use the CHANNEL nn clause to relate the mnemonic name to a particular channel number. The mnemonic name can then be used in a WRITE or SEND statement in place of the phrase CHANNEL nn. The *nn* is an integer from 01 through 99.

Example

```
SPECIAL-NAMES.
    CHANNEL 1 IS CHAN-1.
    .
    .
PROCEDURE DIVISION.
    .
    .
    WRITE...BEFORE ADVANCING CHAN-1
```

ODT Clause

Use the ODT clause to relate a mnemonic name to the Operator Display Terminal. The mnemonic name can then be used in an ACCEPT or DISPLAY statement.

Example

```
SPECIAL-NAMES.
     ODT IS CONSOLE.
     .
     .
     .
 PROCEDURE DIVISION.
     .
     .
     .
     DISPLAY MESSAGE-TEXT UPON CONSOLE.
     ACCEPT MESSAGE-RESPONSE FROM CONSOLE.
```

Switch Clause

The switch clause is used to associate mnemonic names and condition names with program switches. You designate the status of the switches by condition names and you interrogate by testing these names (see "Conditional Expressions" in Section 6). The mnemonic name, if used, is for documentation only; it can never be referred to in the rest of the program. When a mnemonic name is specified for a switch, the compiler prints a message warning that you must not refer to the mnemonic name.

Example

```
SPECIAL-NAMES.
    SW1 IS SPECIAL-REPORT-SWITCH
        ON STATUS IS SW1-ON.
    .
    .
PROCEDURE DIVISION.
    .
    .
    IF SW1-ON
        PERFORM SPECIAL-FUNCTIONS
    ELSE
        PERFORM STANDARD-FUNCTIONS.
```

The switch names are SW1, SW2, SW3, SW4, SW5, SW6, SW7, and SW8. Switches provide a means of communication with the external environment. A switch can have only one of two values, designated as ON and OFF in the switch clause:

- The condition name associated with ON STATUS is TRUE when the switch is set and FALSE when the switch is not set.
- The condition name associated with OFF STATUS is TRUE when the switch is not set and FALSE when the switch is set.

The condition names can be tested at any point in the Procedure Division. For other values for switches see "CALL INTRINSIC Statement" in Section 6.

You can set switches when you initiate the program using the MCP commands VALUE and INSERT. You can also set or reset switches through the ODT during program execution. A switch is set by moving the digit one to its memory location; it is reset by moving the digit zero to its memory location. Switch 8 is located at memory address 0. Switches 1 through 7 are located at memory addresses 1 through 7 respectively.

Note: *SW6 is the object time debugging switch used by the WITH DEBUGGING MODE clause (see "Source-Computer Paragraph," earlier in this section). Therefore, any program compiled with this DEBUGGING MODE clause must have SW6 reserved for debugging.*

Alphabet-Name Clause

The alphabet-name clause provides a means for relating a name to a specified character code set and/or collating sequence. When the alphabet name is referred to in the PROGRAM COLLATING SEQUENCE clause (see "Object-Computer Paragraph," earlier in this section), the alphabet name clause specifies a collating sequence. When the alphabet name is referred to in a CODE-SET clause in a file description entry (see "CODE-SET Clause" in Section 5) the alphabet-name clause specifies a character code set.

ASCII is a synonym for STANDARD-1. If you specify the STANDARD-1 phrase, the character code set and collating sequence identified are those defined in the American National Standard Code for Information Interchange, X3.4–1968.

If you specify the NATIVE phrase, the native character code set and native collating sequence are identified with the alphabet name. The native character code set is the character code set associated with DISPLAY usage, EBCDIC.

The correspondence between characters of the ASCII character code set and characters of the EBCDIC character code set is determined by the Unisys standard tables for translating EBCDIC to ASCII and ASCII to EBCDIC.

EBCDIC and ASCII reference tables appear in Appendix C.

Syntax Rules

The following are the syntax rules for the alphabet-name clause.

- The following are rules for the literals specified in the literal phrase of the alphabet-name clause:
  - If numeric, the literal phrase must be unsigned integers and must have a value within the range of one (1) through the maximum number of characters in the native character set.
  - If nonnumeric and associated with a THROUGH or ALSO phrase, the literal phrase must be one character in length.
- A given character must not be specified more than once in an alphabet-name clause if the literal phrase of the alphabet-name clause is specified.
- The words THRU and THROUGH are equivalent.

General Rules

If you specify the literal phrase, you cannot refer to the alphabet-name in a CODE-SET clause (see "File Description Entry" in Section 5). Define the collating sequence in that phrase according to the following rules:

- If the literal is numeric, the value of each literal designates the ordinal number of a character within the native character set. This value must not exceed the number of characters in the native character set.
- If the literal is nonnumeric, the value of each literal designates the actual character within the native character set. If the value of the nonnumeric literal contains multiple characters, each character of the literal, starting from the leftmost position, is assigned successive ascending positions in the collating sequence.
- The order in which the literals appear in the alphabet-name clause specifies, in ascending sequence, the ordinal number of the character within the collating sequence.

- If any characters within the native collating sequence are not explicitly specified in the literal phrase, they assume in the collating sequence a position greater than that of any of the explicitly specified characters. The relative order within the set of these unspecified characters is unchanged from the order in the native collating sequence.
- If you specify the THROUGH phrase, the set of contiguous characters in the native character set beginning with the character specified by the value of literal-1 and ending with the character specified by the value of literal-2, is assigned a successive ascending position in the collating sequence being defined. In addition, you can designate characters of the native character set to be in either ascending or descending sequence for the set of contiguous characters specified by a given THROUGH phrase.
- If you specify the ALSO phrase, assign the characters of the native character set specified by the value of literal-1, literal-3, literal-4, ... to the same position in the collating sequence.

CURRENCY SIGN Clause

The literal that appears in the CURRENCY SIGN IS clause represents the currency symbol in the PICTURE clause. The literal is limited to a single character that is *not* one of the following:

- Digits 0 through 9
- Alphabetic characters A, B, C, D, I, L, P, R, S, V, X, Z, or the space
- Special characters [*], [+], [−], [,], [.], [;], [(], [)], ["], [/], [=]

If this clause is not present, only the currency sign ($) appears in the PICTURE clause.

DECIMAL POINT IS COMMA Clause

The clause DECIMAL-POINT IS COMMA means that the function of comma and period are exchanged in numeric literals and in the character string of the PICTURE clause.

Input-Output Section

The Input-Output Section deals with the information needed to control transmission of data between external media and the object program. This section is divided into two paragraphs: the File-Control paragraph, which names and associates the files with external media; and the I-O-Control paragraph, which defines special control techniques to be used in the object program.

Note: *If the Input-Output Section header appears in the program, a File-Control paragraph and a SELECT clause must also appear.*

File-Control Paragraph

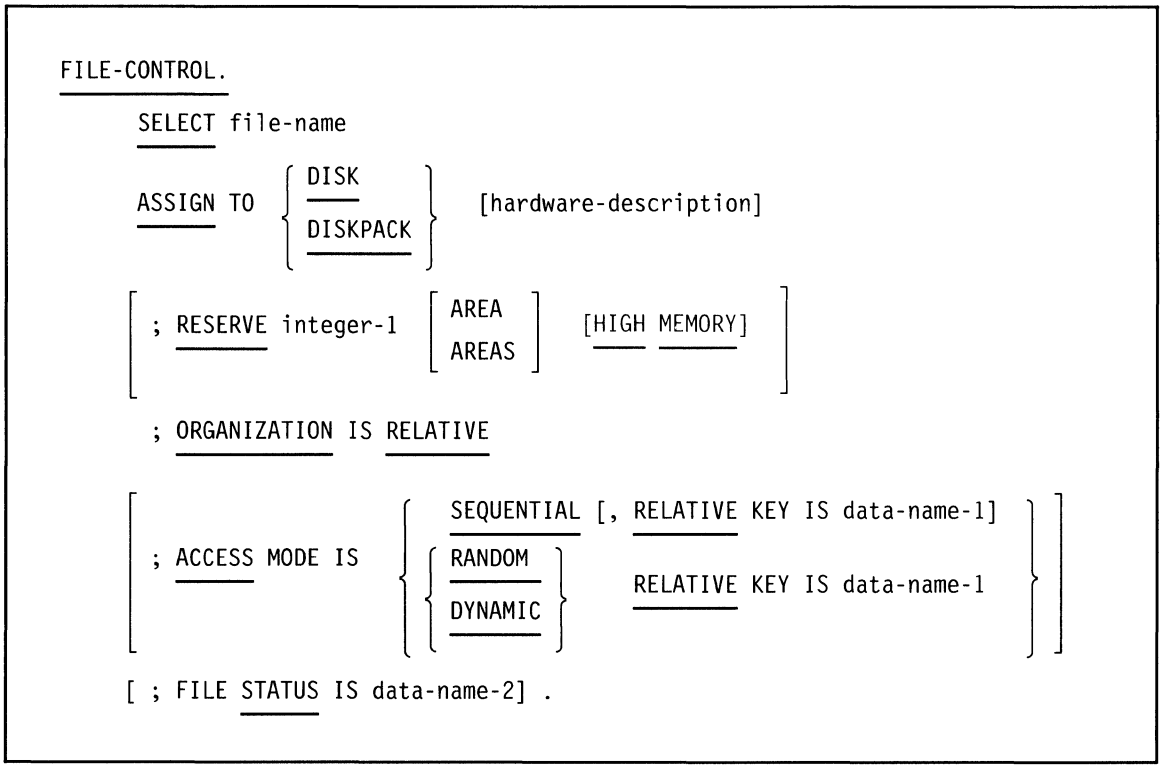
The File-Control paragraph names each file in the program and allows you to specify other file-related information.

The following are the formats:

Format 1 (for sequential files)

```
FILE-CONTROL.
      SELECT [OPTIONAL] file-name
      ASSIGN TO hardware-name [hardware-description]
     [ ; RESERVE integer-1 [ AREA  ]   [HIGH MEMORY] ]
                           [ AREAS ]
     [ ; ORGANIZATION IS SEQUENTIAL]
     [                    { SEQUENTIAL [; ACTUAL KEY IS data-name-1] } ]
     [ ; ACCESS MODE IS   {                                          } ]
     [                    { RANDOM; ACTUAL KEY IS data-name-2        } ]
     [; FILE STATUS IS data-name-3] .
```

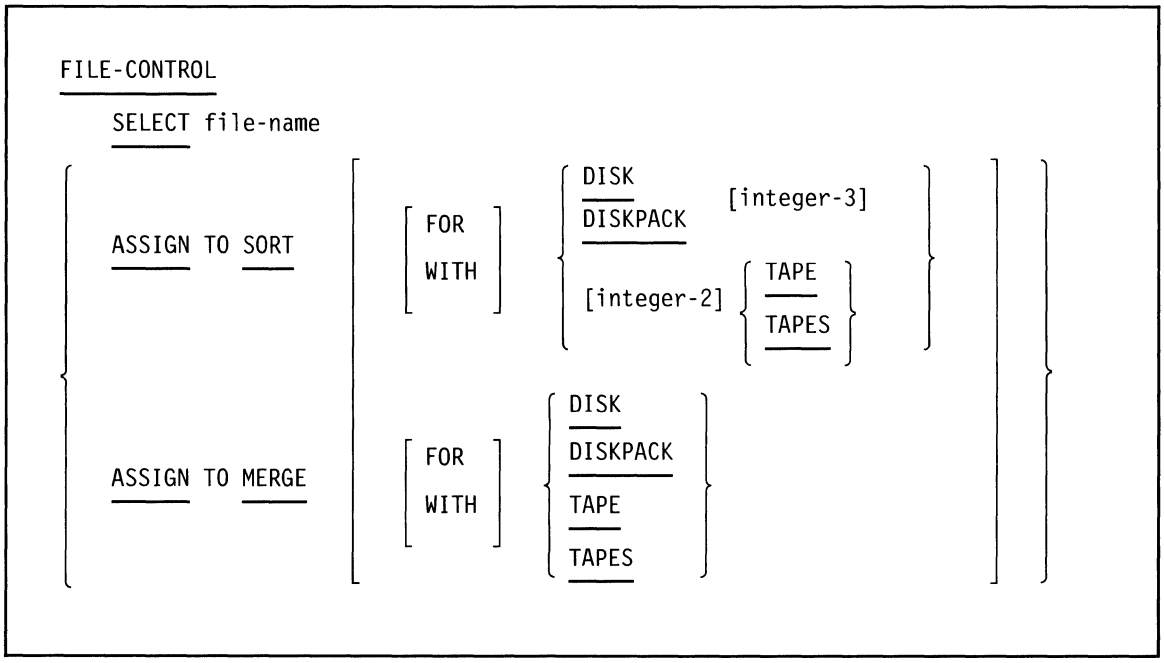
Format 2 (for relative files)



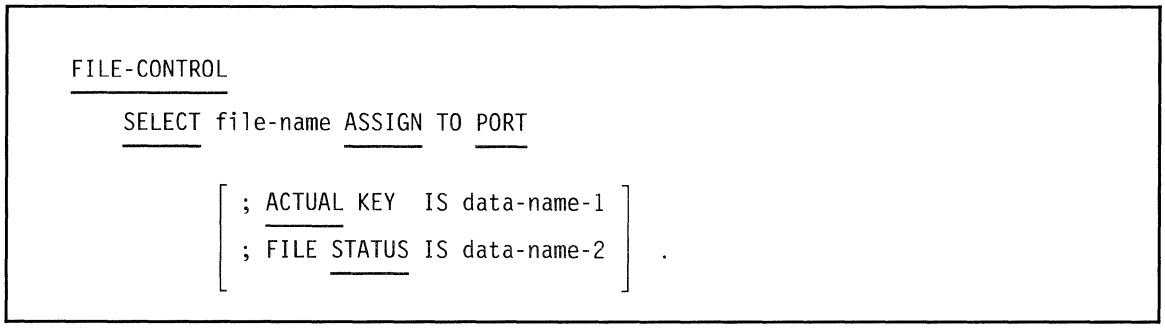
Format 3 (for indexed files)

```
FILE-CONTROL.
      SELECT file-name
      ASSIGN TO { DISK     } [hardware-description]
                { DISKPACK }
    [ ; RESERVE integer-1 [ AREA  ] [HIGH MEMORY] ]
                          [ AREAS ]
      ; ORGANIZATION IS INDEXED
    [                     { SEQUENTIAL } ]
      ACCESS MODE IS      { RANDOM     }
                          { DYNAMIC    }
    [ ; RECORD KEY IS data-name-1
    [ ; ALTERNATE RECORD KEY IS data-name-2  [WITH DUPLICATES] ] ...
    [ ; FILE STATUS IS data-name-3 ] .
```

Format 4 (for sort-merge files)



Format 5 (for port files)



Specify the SELECT clause first in the File-Control entry. File-names used in a program must be the subject of only one SELECT statement. The clauses that follow the SELECT clause can appear in any order. You can qualify data-name-1, data-name-2, and data-name-3.

SELECT Clause

Name each file described in the Data Division using file-name in the File-Control paragraph. Include a file description entry in the Data Division (that is, an entry following the level indicator FD) for each file specified, using Formats 1, 2, or 3 in the file control entry.

Name each sort or merge file described in the Data Division using file-name in Format 4 of the File-Control paragraph. Include a sort-merge file description entry in the Data Division (that is, an entry following the level indicator SD) for each sort or merge file specified using Format 4 in the file control entry.

OPTIONAL Phrase

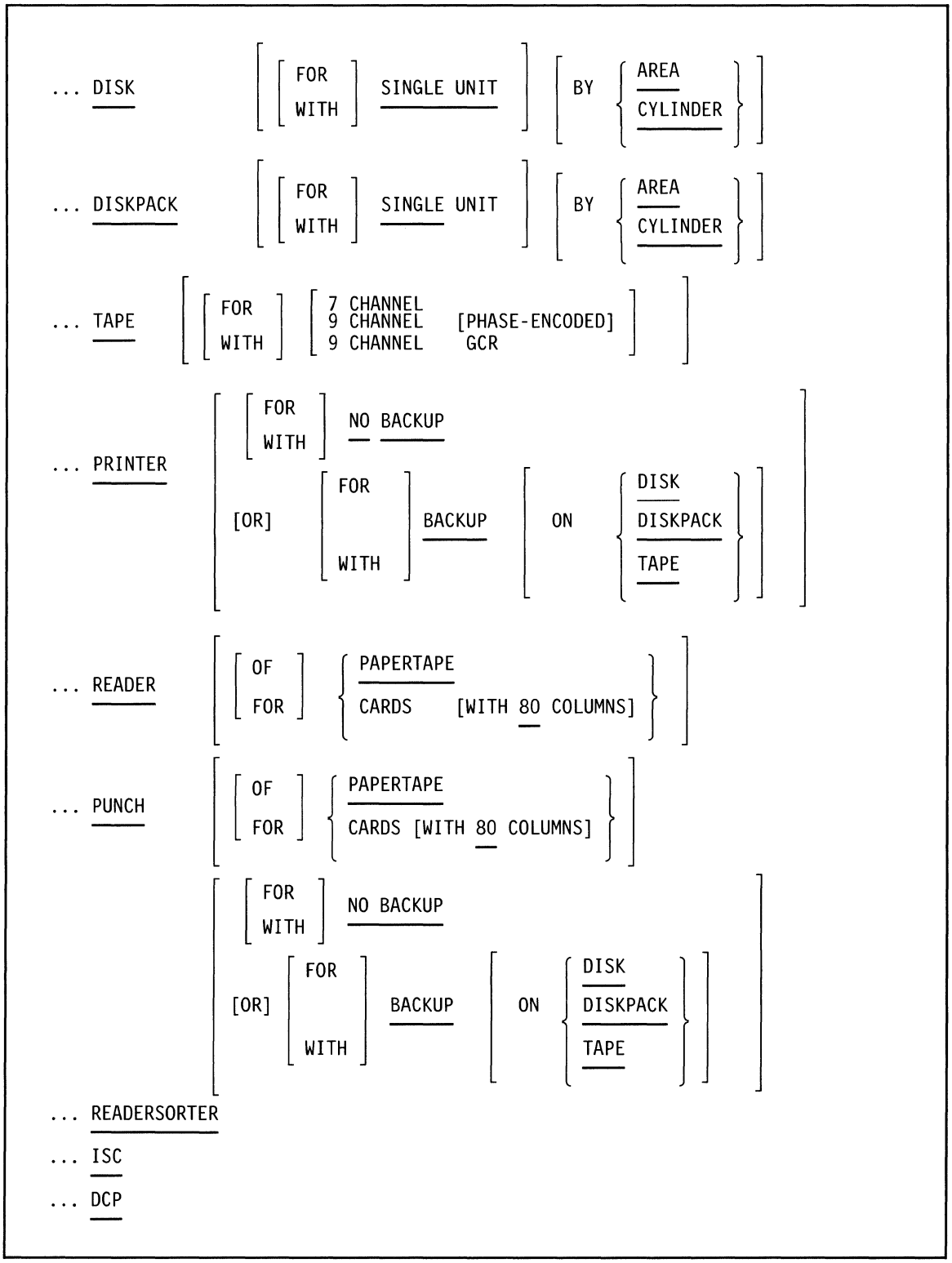
The OPTIONAL phrase can be specified only for input files. Its specification is required for input files that are not necessarily present each time you execute the object program. Refer to "OPEN Statement" in Section 6 for more information on the effect of the OPTIONAL phrase.

ASSIGN Clause

The ASSIGN clause associates the file referred to by file-name with a storage medium.

Format 1

The following are allowable hardware name and hardware description entries for the ASSIGN Clause in Format 1 of the File Control Paragraph. The hardware description portions for DISK and DISKPACK also apply to Formats 2 and 3:


... DISK
[[FOR / WITH] SINGLE UNIT] [BY { AREA / CYLINDER }]
... DISKPACK
[[FOR / WITH] SINGLE UNIT] [BY { AREA / CYLINDER }]
... TAPE
[[FOR / WITH] [7 CHANNEL / 9 CHANNEL / 9 CHANNEL [PHASE-ENCODED] GCR]]
... PRINTER
[[FOR / WITH] NO BACKUP
[OR] [FOR / WITH] BACKUP [ON { DISK / DISKPACK / TAPE }]]
... READER
[[OF / FOR] { PAPERTAPE / CARDS [WITH 80 COLUMNS] }]
... PUNCH
[[OF / FOR] { PAPERTAPE / CARDS [WITH 80 COLUMNS] }]
[[FOR / WITH] NO BACKUP
[OR] [FOR / WITH] BACKUP [ON { DISK / DISKPACK / TAPE }]]
... READERSORTER
... ISC
... DCP

The SINGLE UNIT option forces a DISKPACK file to a single pack. This clause is meaningless for a DISK file.

The AREA option for DISKPACK files causes distribution across available drives on an area basis. This clause is meaningless for a DISK file.

The CYLINDER option forces a DISKPACK file to begin at a cylinder boundary.

If you do not qualify the TAPE assignment, the system assigns any available tape. Specifying 7 CHANNEL allows the system to assign only to a 7-track tape; 9 CHANNEL, to any 9-track tape; 9 CHANNEL PHASE-ENCODED, to a PE tape; and 9 CHANNEL GCR, to a GCR tape.

If you do not specify a device for BACKUP, the system option prevails. The BACKUP option causes PRINTER files or PUNCH files to be placed on a backup tape, disk, or diskpack file, depending on which system option is set or which option the user specified. The NO BACKUP option prevents the file from going to backup automatically when the system option is set and the printer or punch is not available. An operator can manually assign this file to a backup device.

If no clause follows READER or PUNCH, then CARD is assumed. You must state PAPERTAPE if it is desired.

If you assign the file to a READERSORTER, you must also declare a 780-digit buffer area in the record description entry for the file.

You must also initialize the hard interface portion of the buffer. The MCP uses the soft interface area for control and for communicating to the program. For further information see Section 6 "Reader-Sorter Interface" in Volume 2 of this manual.

To implement non-BNA mode Intersystem control (ISC), use the following syntax in the File-Control paragraph:

```
SELECT file-name ASSIGN TO ISC ACCESS RANDOM ACTUAL KEY data name.
```

You must specify a message header containing ISC control commands preceding all text sent through ISC.

When you assign the file to the DCP device, the program can communicate with a B 874, a B 974, an ISC, or a DCDLP. The DCP device has the characteristics of a file whose organization is sequential and access mode is random. The ACTUAL KEY and FILE STATUS clauses have no meaning.

If you specify DCP, allow space for a 40-character header at the beginning of each message. Therefore, the record size specified for the file must be 40 characters larger than the longest message. Also, the file-name, if also used as the external file-name, must match the MCS name specified in the NDL.

Formats 2 and 3

Relative and indexed files can only be assigned to disk or diskpack. The allowable hardware description would be the same as that following DISK and DISKPACK in Format 1.

The compiler assigns key files associated with indexed files to the same hardware device as the data file. It also gives the key files the same values as the data file for the attributes FAMILYNAME, CYLINDERMODE, SINGLEUNIT, SUBSYSTEM, and PROTECTION, as well as all the SECURITY related attributes.

Format 3

If you specify the RESERVE clause for an indexed file, one input/output area is allocated for the data file and the remaining are allocated to be shared by all the key files. The minimum number allowed for the key files is 3 (specified value = 4), and the maximum is 98 (specified value = 99). If you do not specify the RESERVE clause, one input/output area is allocated for the data file, and three are allocated for the key files.

Format 4

Since the file-name represents a sort or merge file, only the ASSIGN clause is permitted to follow the file-name in the File-Control paragraph. The ASSIGN clause specifies the association of the sort or merge file to a storage medium referred to by file-name. When no storage medium is specified, the default is DISK.

Whenever you designate either DISK or DISKPACK, mass storage becomes the primary work medium. For a file assigned to SORT, the MCP disk sort intrinsic *SORT:* is invoked; for a file assigned to MERGE, the MCP intrinsic *MERG:* is invoked.

If you specify TAPE or TAPES as the primary work medium for a SORT file, but do not specify integer-2, the default number of the tapes is three. Integer-2 must have a value within the range of three through eight. The MCP tape sort intrinsic *SORT:* is invoked to accomplish the sorting; the merge intrinsic *MERG:* is invoked to accomplish merging.

If you do not specify hardware type, DISK is assumed.

Integer-3 is a two-digit field used in conjunction with the default media override function of the *SORT:* intrinsic to optionally select a media for sort work files.

For further information on the *SORT:* and *MERG:* intrinsics, refer to the *V Series Systems Operations Guide, Volume 3: System Utilities*.

Format 5

Since file-name represents a port file, only the ASSIGN clause is permitted to follow file-name in the File-Control paragraph.

If you use data-name-1, define it in the DATA DIVISION as an elementary numeric item that describes an unsigned integer.

For port files, the value of the ACTUAL KEY data item specifies the subfile index of the port file. You must specify the ACTUAL KEY clause for a port file that contains more than one subfile. The maximum value permitted for data-name-1 (the maximum number of subfiles for a port) is 9999.

If you use data-name-2, define it in the DATA DIVISION as a two-character data item whose category is alphanumeric. Do not define data-name-2 in the FILE SECTION.

If you specify the FILE STATUS clause, the system moves a value into the item specified by data-name-2 after execution of every statement that explicitly or implicitly refers to this file. This value indicates the status of execution of the statement.

RESERVE Clause

The RESERVE clause allows the user to indicate the number of input/output areas (buffers) to be allocated. A buffer is an area of memory equal in size to a physical block of the file. A file always has a record area (equal in size to a logical record) assigned to it. The buffer areas allocated by the RESERVE clause are in addition to this area. A blocked file must have at least one buffer. If the user designates the RESERVE clause, the number of input/output areas allocated is equal to the value of integer-1. If the RESERVE clause is not specified, the number of input/output areas allocated is 1.

The value of integer-1 can range from 0 through 9. A value of 0 indicates that the file is to have no buffer area. This configuration would normally be useful only in the case of random unblocked files.

HIGH MEMORY Clause

The HIGH MEMORY clause allows you to allocate a file's buffers in the program's HIGH MEMORY areas. Buffers for shared ISAM files are not allowed. Each buffer size is limited to 500KB.

If you do not specify the HIGH MEMORY clause, the buffers are allocated in the program area below the stack.

The HIGH MEMORY clause cannot be used in a program where $DMSICM is set. (For information on dollar options see Section 9).

ORGANIZATION Clause

The ORGANIZATION clause specifies the logical structure of a file. The file organization is established at the time a file is created and cannot subsequently be changed. When you do not specify the ORGANIZATION clause, the ORGANIZATION IS SEQUENTIAL clause is implied and the file is a sequential file.

Requesting ORGANIZATION IS RELATIVE specifies a relative file. Requesting ORGANIZATION IS INDEXED specifies an indexed file.

ACCESS MODE Clause

The ACCESS MODE Clause determines whether the file records are accessed sequentially, randomly, or both ways.

Note that ORGANIZATION and ACCESS MODE have separate and distinct meanings.

If you do not specify the ACCESS MODE clause, the ACCESS MODE IS SEQUENTIAL clause is implied and records can only be accessed sequentially.

For DCP files, ACCESS MODE IS RANDOM is the default value, so you do not need to use the ACCESS MODE clause.

ACCESS MODE IS RANDOM indicates that records can be accessed randomly.

When ACCESS MODE IS DYNAMIC, records in the file can be accessed both sequentially and randomly.

Sequential Files

Records in a sequentially organized file can be accessed in either a sequential or random manner, depending upon the ACCESS MODE clause. You can specify RANDOM as the access mode only for mass storage files. If you do not designate the ACCESS MODE clause, the ACCESS MODE IS SEQUENTIAL clause is implied without the ACTUAL KEY phrase.

When its SHARED attribute is TRUE, a sequential file must have a random access mode and must have an ACTUAL KEY clause.

You can specify the ACTUAL KEY phrase only for mass storage files. Data-name-1 and data-name-2 must be defined in the Data Division as elementary numeric items that describe an unsigned integer PIC 9(8) COMP.

In the sequential access mode, the records are accessed in the ascending sequence of their ordinal location within the file.

In the random access mode, the sequence in which records are accessed is specified by the contents of the ACTUAL KEY data item at the time the program executes the WRITE or READ statement. The value of the ACTUAL KEY data item supplies the ordinal record number of the record to be accessed.

Relative Files

Records in a relative file can be accessed either sequentially or randomly, depending upon the ACCESS MODE clause. If you do not specify the access mode, it is sequential.

When you specify a sequential access mode, the program accesses the records in ascending order by relative record number.

When you specify a random access mode, the programmer controls the sequence in which records are accessed. The programmer accesses the desired record by placing its relative record number in a RELATIVE KEY data item.

When you specify a dynamic access mode, you can change from sequential access to random access, using appropriate forms of input/output statements. (Sequential update operations with DYNAMIC actually use the RANDOM algorithms.)

Indexed Files

Records in an indexed file can be accessed either sequentially or randomly, depending upon the ACCESS MODE clause. If you do not specify the access mode, it is sequential.

When you specify a sequential access mode, the program accesses the records in ascending order by RECORD KEY values. The order of retrieval of records within a key index having duplicate RECORD KEY values is the order in which the records were written into the key index (timewise).

When you specify a random access mode, the programmer controls the sequence in which records are accessed. The programmer accesses the desired record by placing the value of its record key in a RECORD KEY data item.

When you specify a dynamic access mode, you can change from sequential access to random access, using appropriate forms of input/output statements. (Sequential update operations with DYNAMIC actually use the RANDOM algorithms.)

RELATIVE KEY Clause

The following are the rules for the RELATIVE KEY Clause:

1. Specify the RELATIVE KEY clause for a relative file if it is to be referenced by a START statement.

2. Do not define data-name-1 (the RELATIVE KEY) in a record description entry associated with that file-name.
3. Define the data item referenced by data-name-1 as an unsigned integer.
4. Declare the relative key to be no longer than eight digits or characters because only the least significant eight digits or characters of data-name-1 are used as the relative key. Otherwise, a warning will be given.

RECORD KEY and ALTERNATE RECORD KEY Clauses

The following are the rules for the RECORD KEY and the ALTERNATE RECORD KEY clauses:

1. Define the data items referenced by data-name-1 and data-name-2 (the RECORD KEYs) as alphanumeric items within a record description entry associated with that file-name. The data items cannot be longer than 96 characters.
2. Use either data-name-1, data-name-2, or both to describe an item whose size is variable, that is, an item containing an OCCURS clause of various limits.

   Note: *Data-name-2 cannot reference an item whose leftmost character position corresponds to the leftmost character position of an item referenced by data-name-1 or by any other alternate record key associated with this file.*

3. In sequential access mode, records in the file are accessed in the order of ascending record key values within a given key of reference.
4. In random access mode, the value of the RECORD KEY data item indicates the record to be accessed.
5. Use the RECORD KEY clause to specify the prime record key for the file. The value of the prime record key must be unique among records of the file because it provides an access path to records in an indexed file.
6. Use an ALTERNATE RECORD KEY clause to specify an alternate record key for the file. This alternate record key provides an alternate access path to records in an indexed file. You can have a maximum of 98 alternate keys for a file.
7. Use the same data descriptions for data-name-1 and data-name-2, as well as their relative locations within a record, that you used to create the file. If the open type is I/O, then the number of keys declared for the file must also be the same as that used in creating the file. If the open type is INPUT, then you can declare any subset of the originally declared number of alternate keys.
8. Use the DUPLICATES phrase to specify that the value of the associated alternate record key may be duplicated within any of the records in the file. If you do not specify the DUPLICATES phrase, you must not duplicate the value of the associated alternate record key among any of the records in this file.

ACTUAL KEY Clause

If used, data-name-1 must be defined in the DATA DIVISION as an elementary numeric item that describes an unsigned integer.

For port files, the value of the ACTUAL KEY data item specifies the subfile index of the port file. You must specify the ACTUAL KEY clause for a port file that contains more than one subfile. The maximum value permitted for data-name-1 (the maximum number of subfiles for a port) is 9999.

FILE STATUS Clause

When you specify the FILE STATUS clause, a value is moved by the program into the data item specified by data-name-3 in Formats 1 and 3, and by data-name-2 in Format 2. The move occurs after the execution of every statement that references that file either explicitly or implicitly. This value indicates the execution status of the statement.

You must define the FILE STATUS data item in the Data Division as a 2-character alphanumeric data item that can be qualified. Do not define it in the File Section.

Sequential Files

To indicate to the COBOL program the status of that input/output operation, the program places a value in the specified 2-character FILE STATUS data item after executing an OPEN, CLOSE, READ, WRITE, or REWRITE statement and before executing any USE procedure.

Status Key 1

The leftmost character position of the FILE STATUS data item is known as status key 1. It indicates that one of the following conditions exists after the input/output operation is completed:

0 Successful completion. The input/output statement was successfully executed.

1 At end. The sequential READ statement was unsuccessfully executed for one of two reasons:

(1) The program attempted to read a record when no next logical record existed in the file, or
(2) The program executed the first READ statement for a file described with the OPTIONAL clause, but that file was not available to the program at the time its associated OPEN statement was executed.

2 Invalid key. The program executed a Format 2 READ or Format 2 WRITE statement unsuccessfully because the contents of the ACTUAL KEY data item were:

(1) less than 1, or
(2) in the case of a Format 2 READ statement, greater than the ordinal number of the last record written to the file, or
(3) in the case of a Format 2 WRITE statement, greater than the last record allowed to be written because of the specification of a maximum file size.

3 Permanent error. The program tried to execute the input/output statement, but the attempt was unsuccessful because:

(1) there was a boundary violation for a sequential file or
(2) there was an input/output error, such as a data check parity error or a transmission error.

9 Unisys defined condition. The input/output statement encountered conditions other than those already defined and might have been unsuccessfully executed, depending on the value of status key 2.

Status Key 2

The rightmost character position of the FILE STATUS data item is known as status key 2. It is used to describe further the results of the input/output operation. This character contains a value as follows:

- When no further information is available concerning the input/output operation, then status key 2 contains a value of 0.
- When status key 1 contains a value of 3, status key 2 may contain a value of 4 indicating a boundary violation. This condition indicates that the program attempted to write beyond the externally defined boundaries of a sequential file. These boundaries are defined by the file attributes AREA and AREASIZE (see "VALUE OF Clause" in Section 5).
- When status key 1 contains a value of 9 indicating a Unisys defined condition, the value of status key 2 indicates the name of the condition as follows:

  1 Short block. Because of the limitation of the physical recording medium, the system cannot determine whether the logical record returned was written to the file. The programmer must determine the validity of the data record.

2 Data error. When the programmer declares the logical records to be variable in length and supplies the logical record length (via the RECORD CONTAINS clause), a data error occurs on a Read, Write, or Rewrite statement if the logical record length supplied is less than the minimum record size or greater than the maximum record size declared for the file. This condition neither initiates an input/output operation nor causes data to be transferred to or from the record area.

6 Timeout. A time limit elapsed prior to the transfer of data to or from the hardware device.

7 Break on output. For an output or input/output file, this condition occurs if the physical hardware device is equipped with a break that allows the operator to halt the transfer of data in process.

9 Unexpected I/O error. An error of undeterminable nature might have occurred in the input/output operation.

Valid Combinations of Status Keys 1 and 2

Table 4–1 shows the valid permissible combinations of the values of status keys 1 and 2.

Table 4–1. Valid Combinations of Status Keys 1 and 2 for Sequential Files

Key 1	Key 2	Exception
0	0	Function was successful
1	0	End of file encountered
2	0	Invalid key
3	0	Permanent I/O error
3	4	Boundary violation
9	1	Short block
9	2	Data error
9	6	Timeout
9	7	Break on output
9	9	Unexpected I/O error

Relative Files

If you specify the FILE STATUS clause in a file control entry, the program places a value into the specified 2-character data item when an OPEN, CLOSE, READ, WRITE, REWRITE, DELETE, or START statement is executed and before any

applicable USE procedure is executed. This value indicates to the COBOL program the status of that input/output operation.

Status Key 1

The leftmost character position of the FILE STATUS data item is known as status key 1. It is set to indicate one of the following conditions after an input/output operation is completed:

0 Successful completion. The input/output statement was successfully executed.

1 At end. The Format 1 READ statement was unsuccessfully executed because the program attempted to read a record when no next logical record existed in the file.

2 Invalid key. The input/output statement was unsuccessfully executed for one of the following reasons:

Duplicate key
No record found
Boundary violation

9 Unisys defined condition. The input/output statement was unsuccessfully executed because of a condition specified by the implementer. This value is used only to indicate a condition not indicated by other defined values of status key 1 or by specified combinations of the values of status key 1 and status key 2.

F Unisys defined fatal error. The input/output statement was unsuccessfully executed because of a condition specified by the implementer. The program is aborted.

Status Key 2

The rightmost character position of the FILE STATUS data item is known as status key 2. It is used to further describe the results of the input/output operation. This character contains a value as follows:

- When no further information is available concerning the input/output operation, then status key 2 contains a value of 0.
- When status key 1 contains a value of 2 indicating an INVALID KEY condition, status key 2 is used to designate the cause of that condition, as follows:
  - A value of 2 indicates a duplicate key value. The program attempted to write a record that would create a duplicate key in a relative file.
  - A value of 3 indicates no record was found. The program attempted to access a record identified by a nonexistent key value.
  - A value of 4 indicates a boundary violation. The program attempted to write beyond the externally defined boundaries of a relative file.

- When status key 1 contains a value of 9 or F, indicating a Unisys-defined condition, the value of status key 2 is as defined in Table 4–2.

Valid Combinations of Status Keys 1 and 2 - Relative Files

Table 4–2 shows the valid combinations of positions 1 and 2 of the 2-character status key for relative files and the exceptions they indicate.

Table 4–2 . Valid Combinations of Status Keys 1 and 2 for Relative Files

Key 1	Key 2	Exception
0	0	Function was successful.
1	0	End of file was encountered.
2	2	Record with this key value is already present.
2	3	No record with this key value was found.
2	4	File full limit error occurred.
9	1	Open type is incompatible with requested function.
9	2	Current record is not defined.
9	4	Last function was not a successful READ operation.
9	5	New record length differs from old on REWRITE operation (variable length records only).
9	6	Indexed file in use or requires recovery.
F	6	File declaration did not match creation attributes.
F	9	Cannot open or close a file that is open or closed.

When the intrinsic encounters an F in Key 1, a message is displayed on the ODT, the program memory is dumped, and the program is aborted. The message includes the nature of the error and the value of the status digits. Errors with status digit values of 1:0 through 9:6 can also be fatal if an exception routine has not been declared.

The following are fatal error messages for relative file handling:

<file-name>	:READ/WRITE EOF ERROR	(ERR=F0)
	:FILE ATTRIBUTE MISMATCH	(ERR=F6)
	:ALREADY OPENED/CLOSED	(ERR=F9)
	:NO USER EXCEPTION ADDRESS	(ERR=xx)

Indexed Files

If you specify the FILE STATUS clause in a file control entry, the program places a value into the specified 2-character data item when an OPEN, CLOSE, READ, WRITE, REWRITE, DELETE, or START statement is executed and before any

applicable USE procedure is executed. This value indicates to the COBOL program the status of that input/output operation.

Status Key 1

The leftmost character position of the FILE STATUS data item is known as status key 1. It is set to indicate one of the following conditions after an input/output operation is completed:

0 Successful completion. The input/output statement was successfully executed.

1 At end. The Format 1 READ statement was unsuccessfully executed because the program attempted to read a record when no next logical record existed in the file.

2 Invalid key. The input/output statement was unsuccessfully executed for one of the following reasons:

Sequence error
Duplicate key
No record found
Boundary violation

9 Unisys defined condition. The input/output statement was unsuccessfully executed because of a condition specified by the implementer. This value is used only to indicate a condition not indicated by other defined values of status key 1 or by specified combinations of the value of status key 1 and status key 2.

F Unisys defined fatal error. The input/output statement was unsuccessfully executed because of a condition specified by the implementer. The program is aborted.

Status Key 2

The rightmost character position of the FILE STATUS data item is known as status key 2. It is used to describe the results of the input/output operation. This character contains a value as follows:

- When no further information is available concerning the input/output operation, then status key 2 contains a value of 0.
- When status key 1 contains a value of 0 indicating a successful completion, status key 2 might contain a value of 2 indicating a duplicate key. This condition indicates the following:
  - For a READ statement, the key value for the current key of reference is equal to the value of that same key in the next record within the current key of reference.
  - For a WRITE or REWRITE statement, the record just written created a duplicate key value for at least one alternate record key for which duplicates are allowed.

- When status key 1 contains a value of 2 indicating an INVALID KEY condition, status key 2 is used to designate the cause of that condition, as follows:
  - A value of 1 indicates a sequence error for a sequentially accessed indexed file. The ascending sequence requirements of successive record key values have been violated (see the "WRITE Statement" in Section 6), or the prime record key value has been changed by the COBOL program between executions first of a READ statement and then of the next sequential REWRITE or DELETE statement for that file.
  - A value of 2 indicates a duplicate key value. The program attempted to write or rewrite a record that would create a duplicate key in an indexed file.
  - A value of 3 indicates no record found. The program attempted to access a record identified by a key, but that record does not exist in the file.
  - A value of 4 indicates a boundary violation. The program attempted to write beyond the externally defined boundaries of an indexed file.
- When status key 1 contains a value of 9 or F, indicating a Unisys defined condition, the value of status key 2 is as defined in Table 4–3.

Valid Combinations of Status Keys 1 and 2 - Indexed Files

Table 4–3 shows the valid combinations of positions 1 and 2 of the 2-character status key for indexed files and the exceptions they indicate.

Table 4–3. Valid Combinations of Status Keys 1 and 2 for Indexed Files

Key 1	Key 2	Exception
0	0	Function was successful (ALL).
0	2	- Next record has duplicate key value (READ). - Duplicate key entry was found during key file insertion (WRITE, REWRITE, DELETE-RECOVERY).
1	0	-Logical EOF was exceeded (SEQ READ). - File is empty (READ, START).
2	1	Prime key was changed between last READ and this function (SEQ REWRITE).
2	2	Duplicate key entry was found, but not allowed on this key (WRITE, REWRITE, DELETE-RECOVERY).
2	3	- No record was found in file with this key (RAN READ, START=, REWRITE, DELETE). - No record was found in file with key > or >= this key (START>, START>=). - Logical EOF was exceeded (> EOF, = 0) (START, REWRITE, DELETE).

Table 4–3. Valid Combinations of Status Keys 1 and 2 for Indexed Files (cont.)

Key 1	Key 2	Exception
2	4	- Data file physical EOF was exceeded (WRITE). - Key file physical EOF was exceeded during table split (WRITE, REWRITE, DELETE-RECOVERY).
9	1	OPEN type is incompatible with requested function: - READ with file NOT OPEN or OPEN OUTPUT - START with file NOT OPEN or OPEN OUTPUT - WRITE with file NOT OPEN or OPEN OUTPUT - REWRITE with file NOT OPEN or OPEN INPUT or OUTPUT - DELETE with file NOT OPEN or OPEN INPUT or OUTPUT
9	2	Current record was not defined (SEQ READ).
9	3	Current key of reference is undefined (= 0, > number keys) (READ, START).
9	4	Last function was not a successful sequential READ (SEQ REWRITE, DELETE).
9	5	New record length of variable length records differs from old, so operation is aborted (REWRITE).
9	6	File is in use by updating program (OPEN INPUT, IO).
9	7	File recovery is required because the previous update program was terminated prior to completing the update (OPEN IN,IO).
F	0	Unexpected access routine EOF (block > MAX block, = 0) occurred on data or key file (READ, START, WRITE, REWRITE, DELETE).
F	1	IF BLOCK smashed, IF BLOCK FD did not match FD in the CONTROL BLOCK (CLOSE).
F	2	File is corrupted: - Search key > highest table entry (RAN READ, START, RAN REWRITE, RAN DELETE). - Search key didn't match key in retrieved record (READ, RAN REWRITE, RAN DELETE). - Key file entry pointed to empty data record slot (READ, RAN REWRITE, RAN DELETE).
F	3	File is corrupted because update function in process failed and could not be recovered (WRITE, REWRITE, DELETE).
F	4	Data file is corrupted because data block in free space list had no empty record slots (WRITE).
F	5	File versions are mismatched because key file date/time stamp did not match data file date/time stamp (OPEN).

Table 4–3. Valid Combinations of Status Keys 1 and 2 for Indexed Files (cont.)

Key 1	Key 2	Exception
F	6	Data or prime key file attribute is mismatched because IF BLOCK FD did not match CONTROL BLOCK FD (OPEN INPUT,IO).
F	7	Alternate key declaration is missing or mismatched: - Number keys in IF BLOCK differs from CTL BLK (OPEN IO) - IF BLOCK FD unmatched in CONTROL BLK (OPEN IN, IO)
F	9	- CLOSE was attempted on a file that was not OPEN. - OPEN was attempted on a file that was already OPEN.
F	A	Unexpected buffer is locking deadlock (WRITE, REWRITE, DELETE-RECOVERY).

When the intrinsic encounters an F in Key 1, a message is displayed on the ODT, the program memory is dumped, and the program is aborted. The message includes the nature of the problem and the value of the status digits. Errors with status digit values of 1:0 through 9:6 can also be fatal if an exception routine has not been declared.

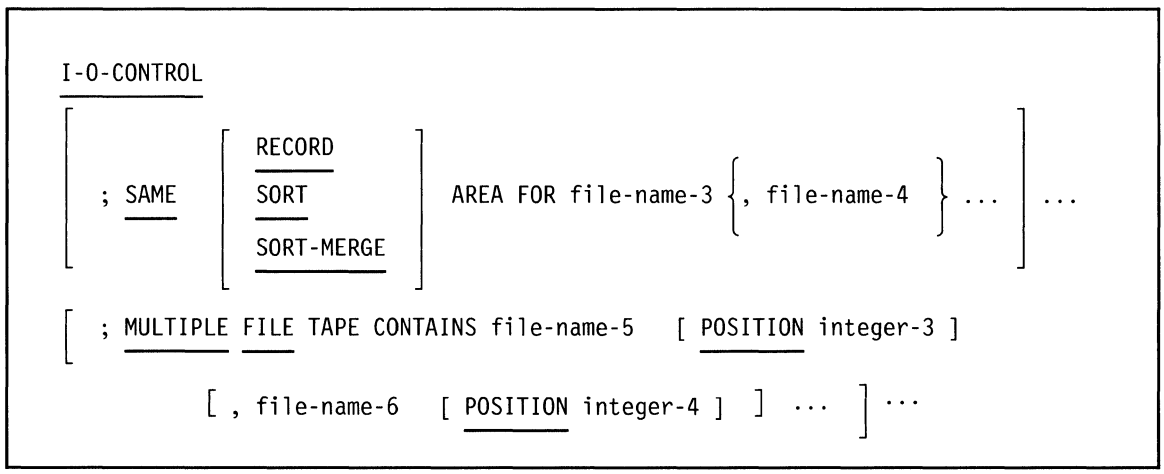
The following are fatal error messages for indexed file handling:

<file-name>	:READ/WRITE EOF ERROR	(ERR=F0)
	:IF/RF BLOCK SMASHED	(ERR=F1)
	:INDEXED FILE CORRUPTED	(ERR=F2)
	:UPDATE RECOVERY FAILURE	(EFF=F3)
	:FREE BLOCK LIST ERROR	(ERR=F4)
	:DATE/TIME STAMP MISMATCH	(ERR=F5)
	:FILE ATTRIBUTE MISMATCH	(ERR=F6)
	:MISSING KEY DECLARATION	(ERR=F7)
	:ALREADY OPENED/CLOSED	(ERR=F9)
	:BUFFER HANDLING DEADLOCK	(ERR=FA)
	:NO USER EXCEPTION ADDRESS	(ERR=xx)

I-O-Control Paragraph

The I-O-Control paragraph specifies the memory area that is to be shared by different files and the location of files on a multiple file reel.

The following is the general format:



The I-O-Control paragraph is optional.

SAME Clause

The SAME AREA clause specifies that two or more files not representing sort or merge files are to use the same memory area during processing. The shared area includes all storage assigned to the files specified (both record and buffer areas). Therefore, more than one of the files cannot be open at the same time.

The SAME RECORD AREA clause specifies that two or more files are to use the same memory area to process the current logical record. All the files can be open simultaneously. A logical record in the SAME RECORD AREA is considered a logical record of each open input file and open output file whose file-name appears in this SAME RECORD AREA clause.

This is equivalent to an implicit redefinition of the area; that is, records are aligned on the leftmost character position.

Files referenced in the SAME AREA, SAME RECORD AREA, SAME SORT AREA, or SAME SORT-MERGE AREA clause need not have the same organization or access. SAME SORT AREA and SAME SORT-MERGE AREA are equivalent.

More than one SAME clause may be included in a program, with the following constraints:

- A file-name must not appear in more than one SAME AREA clause.

- A file-name must not appear in more than one SAME RECORD AREA clause.
- If one or more file-names of a SAME AREA clause appear in a SAME RECORD AREA clause, all file-names in that SAME AREA clause must appear in the SAME RECORD AREA clause. However, additional file-names not appearing in that SAME AREA clause can also appear in that SAME RECORD AREA clause. The rule that only one of the files mentioned in a SAME AREA clause can be open at any given time takes precedence over the rule that all files mentioned in a SAME RECORD AREA clause can be open at any given time.
- A file-name that represents a sort or merge file must not appear in more than one SAME SORT AREA or SAME SORT-MERGE AREA clause.
- If a file-name that does not represent a sort or merge file appears in a SAME AREA clause and one or more SAME SORT AREA or SAME SORT-MERGE AREA clauses, all of the files named in the SAME AREA clause must be named in the SAME SORT AREA or SAME SORT-MERGE AREA clause(s).

If you use the SAME SORT AREA or SAME SORT-MERGE AREA clause, at least one file-name must represent a sort or merge file. Files that do not represent sort or merge files can also be named in the clause. This clause specifies that storage is shared as follows:

- A memory area is made available for use in sorting or merging each sort or merge file named. Thus any memory area allocated for this sorting or merging is available for reuse for other sort or merge files.
- You can allocate storage areas assigned to files that do not represent sort or merge files, as needed for sorting or merging the sort or merge files named in the SAME SORT AREA or SAME SORT-MERGE AREA clause.
- Files other than sort or merge files do not share the same storage area. If you want these files to share the same storage area, you must also include in the program a SAME AREA or SAME RECORD AREA clause.
- Any non-sort-merge files named in this clause must not be open during the execution of a SORT or MERGE statement that refers to a sort or merge file it names.

MULTIPLE FILE Clause

The MULTIPLE FILE clause is required when more than one file shares the same physical reel of tape. Only those files that are used in the object program need be specified. If you list all file names in consecutive order, you need not give the POSITION clause. If any file in the sequence is not listed, you must give the position relative to the beginning of the tape. Not more than one file on the same tape reel can be open at one time.

Coding Example

Example 4–1 shows an example of the Environment Division coding.

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.  V-300 , WITH DEBUGGING MODE.
OBJECT-COMPUTER.  V-300 , SEQUENCE COL-SEQ.
SPECIAL-NAMES.  SW1  ON IS TRUE-TEST, OFF IS FALSE-TEST,
    COL-SEQ IS ASCII.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT OPTIONAL INPUT-FILE ASSIGN TO READER.
    SELECT INPUT-2 ASSIGN TO TAPE 9 CHANNEL PHASE-ENCODED.
    SELECT SORT-FILE ASSIGN TO SORT FOR DISK.
    SELECT WORK-FILE ASSIGN TO DISKPACK BY AREA; RESERVE 10;
        ORGANIZATION SEQUENTIAL;  ACCESS RANDOM; ACTUAL KEY KEY-REC;
        FILE STATUS IS TWO-KEY.
I-O-CONTROL.
    SAME AREA INPUT-FILE, INPUT-2.
```

Example 4–1. Coding the Environment Division

Section 5
Data Division

This section describes the elements that can be included in the third division of a COBOL program, the Data Division.

General

The Data Division describes the data that the object program is to manipulate, create, accept as input, or produce as output. Data to be processed falls into three categories:

- That which is contained in files and enters or leaves the internal memory of the computer from a specified area or areas
- That which is developed internally and placed into intermediate or Working-Storage
- Constants that are defined by the user

The Data Division is one of the required divisions in a COBOL program. It consists of a header, *DATA DIVISION*, followed by three optional sections: the File Section, the Working-Storage Section, and the Linkage Section.

The File Section defines the structure of data files. Each file is defined by a file description entry and one or more record descriptions. Include record descriptions immediately following the file description entry.

The Working-Storage Section describes records and noncontiguous data items that are not part of external data files but are developed and processed internally. It also describes data items assigned in the source program that do not change when the object program is executed.

The Linkage Section describes data that is used for inter-program communication.

Structure

Figure 5-1 shows the general structure of the Data Division:

```
DATA DIVISION.
[ FILE SECTION.                                          ]
[     [ file-description-entry                           ]
[       {record-description-entry} ...                   ]
[                                                        ]
[     [ sort-merge-file-description-entry                ]
[       {record-description-entry} ...   ] ...           ]

[ WORKING-STORAGE SECTION.                               ]
[   [77-level-description-entry] ...                     ]
[                                                        ]
[   [record-description-entry] ...                       ]

[ LINKAGE SECTION.                                       ]
[   [77-level-description-entry] ...                     ]
[                                                        ]
[   [record-description-entry] ...                       ]
```

Figure 5–1. Structure of the Data Division

Records

The record description consists of a set of data description entries that describe the characteristics of a particular record. Each data description entry consists of a level number followed by a data name, if required, and by a series of independent clauses, as required.

Levels

A level concept is inherent in the structure of a logical record because subdivisions of a record must be specified for the purpose of data reference. Once a subdivision has been specified, it can be further subdivided to permit more detailed data referral.

The most basic subdivisions of a record are called elementary items. Consequently, a record either consists of a sequence of elementary items, or is an elementary item itself.

In order to refer to a set of elementary items, the elementary items are combined into groups. Each group item consists of a named sequence of one or more elementary items. Groups, in turn, can be combined into groups of two or more, and so on. Thus, an elementary item can belong to more than one group.

Level Numbers

A system of level numbers shows the organization of elementary items and group items. A group includes all group and elementary items that follow it, up to the next level number that is less than or equal to the level number of that group. All items that are immediately subordinate to a given group item must be described, using identical level numbers greater than the level number used to describe that group item.

Since records are the most inclusive data items, level numbers for records start at 01. Less inclusive data items are assigned higher level numbers, but not greater in value than 49. Special level numbers 66, 77, and 88 are exceptions to this rule. Separate entries are written in the source program for each level number used.

The special level numbers 66, 77, and 88 represent entries for which there is no true concept of level. Level number 66 is assigned to entries describing items by means of RENAMES clauses for the purpose of regrouping data items. Level number 77 is assigned to entries that specify noncontiguous data items, which are not subdivisions of other items and are not themselves subdivided. Level number 88 is assigned to entries that specify condition names to particular values of a conditional variable.

Figure 5–2 illustrates a record layout that might occur in a business environment. Example 5–1 shows how this record could be described in COBOL.

80 COLUMN CARD INPUT

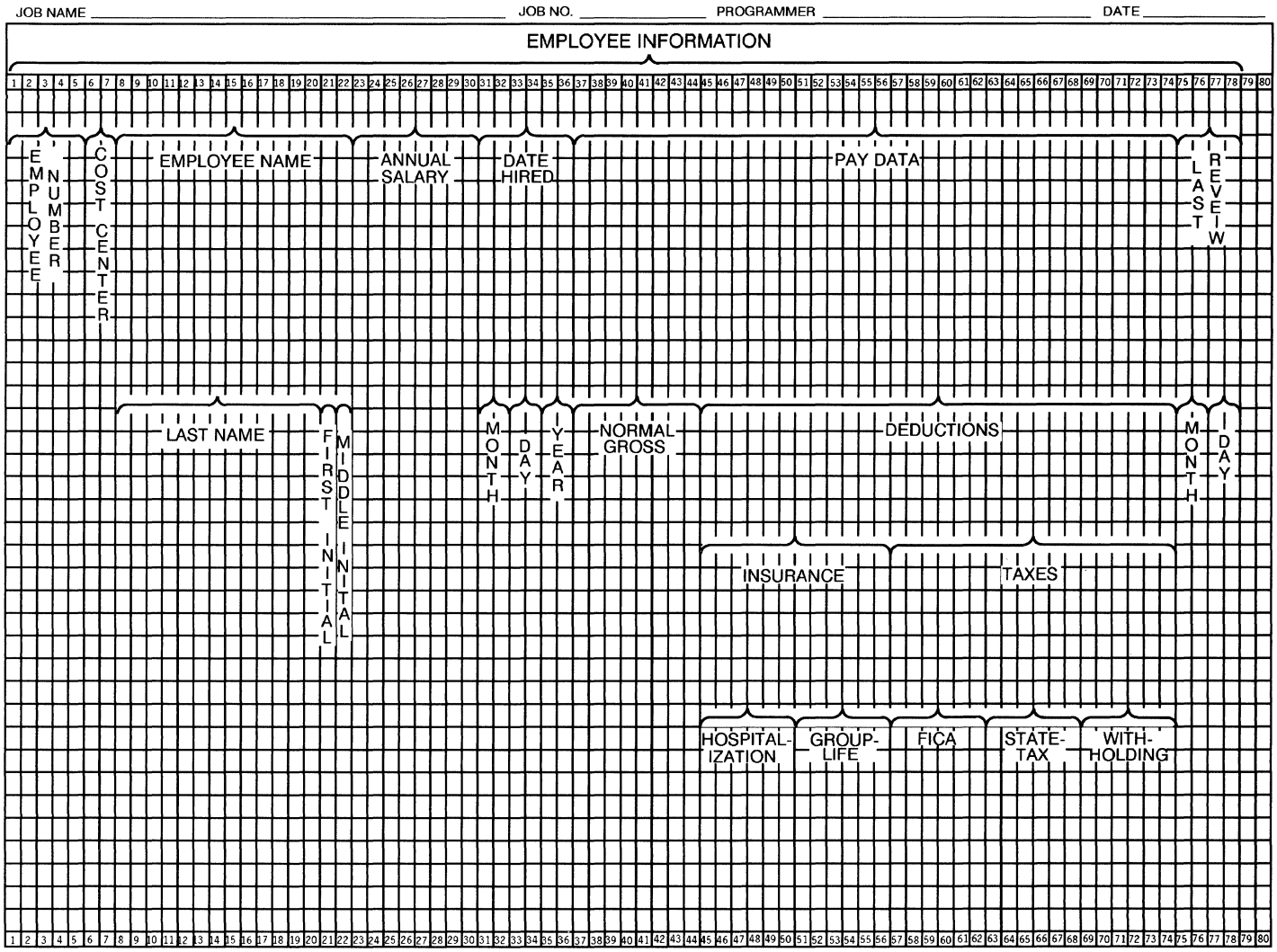


Figure 5–2. Example of a Record Format

```
01  EMPLOYEE-INFO.
    03        EMP-NO                PIC 9(5).
    03        EMP-COST-CNTR         PIC 99.
    03        EMP-NAME.
      05      EMP-LAST-NAME         PIC X(13).
      05      EMP-FIRST-INITIAL     PIC X.
      05      EMP-M-INITIAL         PIC X.
    03        EMP-ANNUAL-SALARY     PIC 9(6)V99.
    03        EMP-DT-HIRED.
      05      EMP-H-MONTH           PIC 99.
      05      EMP-H-DAY             PIC 99.
      05      EMP-H-YEAR            PIC 99.
    03        EMP-PAY-DATA
      05      EMP-GROSS             PIC 9(6)V99.
      05      EMP-DEDUCTIONS.
        07    EMP-INSURANCE.
          09  EMP-HOSPITAL          PIC 9(4)V99.
          09  EMP-LIFE              PIC 9(4)V99.
        07    EMP-TAXES.
          09  EMP-FICA              PIC 9(4)V99.
          09  EMP-STATE-TAX         PIC 9(4)V99.
          09  EMP-WITHHOLDING       PIC 9(4)V99.
    03        EMP-LAST-REVIEW.
      05      EMP-R-MONTH           PIC 99.
      05      EMP-R-DAY             PIC 99.
```

Example 5–1. Level Number Construction for Record in Figure 5–2

In Figure 5–2 and Example 5–1, EMP-NO, EMP-COST-CENTER, EMP-LAST-NAME, EMP-FIRST-INITIAL, EMP-M-INITIAL, EMP-ANNUAL-SALARY, EMP-H-MONTH, EMP-H-DAY, EMP-H-YEAR, EMP-GROSS, EMP-HOSPITAL, EMP-LIFE, EMP-FICA, EMP-STATE-TAX, EMP-WITHOLDING, EMP-R-MONTH, and EMP-R-DAY are all elementary items. Each elementary item belongs to one or more groups.

In the example, EMP-HOSPITAL is part of the EMP-INSURANCE group. EMP-INSURANCE, in turn, is a part of the EMP-DEDUCTIONS, which is part of the EMP-PAY-DATA group. EMP-PAY-DATA group includes all items up to but not including EMP-LAST-REVIEW (which has an equal level number). The EMP-DEDUCTIONS group includes all items up to but not including EMP-LAST-REVIEW (which has a level number less than that of EMP-DEDUCTIONS).

Data

Data concepts include data classification, categorization, qualification to ensure uniqueness, and alignment (see Table 5–1).

Data Classification

There are five categories of data items: alphabetic, numeric, numeric-edited, alphanumeric-edited, and alphanumeric. These five categories are grouped into three classes: alphabetic, numeric, and alphanumeric. For alphabetic and numeric, the classes and categories are synonymous. The alphanumeric class includes the categories of alphanumeric-edited, numeric-edited, and alphanumeric (without editing). Every elementary item except for an index data item belongs to one of the classes and further to one of the categories. The class of a group item is treated at object time as alphanumeric regardless of the class of elementary items subordinate to that group item. Table 5–1 depicts the relationship of the classes and categories of data items.

Table 5–1. Classes and Categories of Data Items

Level of Item	Class	Category
Elementary	Alphabetic	Alphabetic
	Numeric	Numeric
	Alphanumeric	Numeric Edited Alphanumeric Edited Alphanumeric
Group	Alphanumeric	Alphabetic Numeric Numeric Edited Alphanumeric Edited Alphanumeric

Unique Data References

An identifier is a term used throughout this manual to indicate that a data name, if not unique, must be followed by a syntactically correct combination of qualifiers necessary to ensure uniqueness. Identifiers can be subscripted or indexed unless otherwise stated.

Qualification

Every user-specified name that defines an element in a COBOL source program must be unique, either because no other name has the identical spelling and hyphenation or because the name is made unique by mentioning it with one or more of the higher levels of its hierarchy. The higher levels are called qualifiers and this process that specifies uniqueness is called qualification. Enough qualification must be mentioned to make the name unique; however, it may not be necessary to mention all levels of the hierarchy.

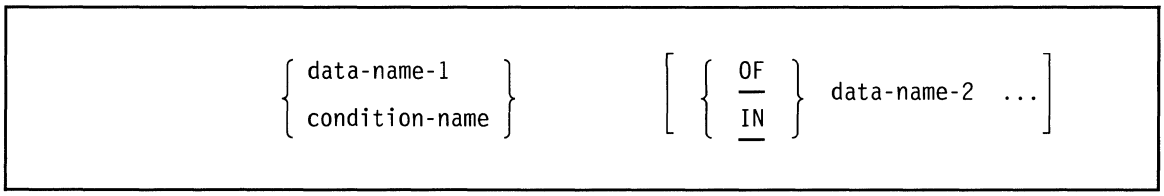
Within the Data Division, all data names used for qualification must be associated with a level indicator or a level number. Therefore, two identical data names must not appear as entries subordinate to a group item unless they can be made unique through qualification.

In the hierarchy of qualification, names associated with a level indicator (such as FD) are more significant than names associated with level number 01, which are more significant than names associated with level number 02, and so on through 49. Thus, the most significant name in the hierarchy must be unique and cannot be qualified.

Subscripted or indexed data names and conditional variables, as well as data names, can be made unique by qualification. The name of a conditional variable can be used as a qualifier for any of its condition names. Regardless of the available qualification, no name can be both a data name and a procedure name.

Qualification is performed by following a data name or condition name with one or more phrases composed of a qualifier preceded by IN or OF. IN and OF are logically equivalent.

The following is the general format for data name qualification:



The following are the rules for data name qualification:

- Put each qualifier at a successively higher level and within the same hierarchy as the name it qualifies.
- Do not use the the same name at two levels in a hierarchy.
- If you assign a data name or a condition name to more than one data item in a source program, qualify the data name or condition name each time it is referred to in the Procedure, Environment, and Data Divisions (except in the REDEFINES clause where qualification is unnecessary and must not be used).
- Do not subscript a data name when it is used as a qualifier.
- You can qualify a name even though it does not need qualification. If there is more than one combination of qualifiers that ensures uniqueness, then any such set can be used. The complete set of qualifiers for a data name must not be the same as any partial set of qualifiers for another data name.
- When you qualify a subscripted or indexed data name, you must put the qualification before the subscript.

Data Alignment

Positioning of data within an elementary item depends on the category of the receiving item. The rules are the following:

- If the receiving data item is described as numeric
  - The data is aligned by decimal point and is moved to the receiving character positions with zero fill or truncation at either end, as required.
  - When an assumed decimal point is not explicitly specified, the data item is treated as if it had an assumed decimal point immediately following its rightmost character. It is then aligned as described in the preceding paragraph.
- If the receiving data item is a numeric-edited data item, the data moved to the edited data item is aligned by decimal point with zero fill or truncation at either end, as required. The leading zeros may be replaced because of editing requirements.
- If the receiving data item is alphanumeric (other than a numeric-edited data item), alphanumeric-edited or alphabetic, the sending data is moved to the receiving character positions, aligned at the leftmost character position in the data item, and has either space fill or truncation to the right, as required.

If you specify the JUSTIFIED clause for the receiving item, these standard rules are modified; see "JUSTIFIED clause" in this section.

Tables

In COBOL, you can define tables of contiguous data items and access an item relative to its position in the table. Constructs are provided for specifying how many times an item is to be repeated. Each item can be identified through use of a subscript or an index.

COBOL provides access to items in multi-dimensional variable length tables and facilities for specifying ascending or descending keys. The language also permits searching a dimension of a table for an item to satisfy a specified condition.

In COBOL, you define tables composed of contiguous data items by including the OCCURS clause in the data description entries. This clause specifies the number of times that the item is to be repeated. The item is considered a table element, and its name and description apply to each repetition or occurrence. Since the occurrence of each table element is assigned a unique data name, you can refer to a desired occurrence only by specifying the data name of the table element together with the occurrence number of the element. Subscripting and indexing are the methods used to specify the occurrence number of a desired table element.

Table Definition

To define a one-dimensional table, the programmer uses an OCCURS clause as part of the data description of the table element, but the OCCURS clause must not appear in the description group items that contain the table element.

Example 5–2 shows the syntax for a one-dimensional table.

```
01  TABLE-1.
    03  TABLE-ELEMENT   OCCURS  20 TIMES
        05  NAME         PIC X(30).
        05  SSAN         PIC 9(6).
```

Example 5–2. One-Dimensional Table

Defining a one-dimensional table within occurrences of elements of another one-dimensional table creates a two-dimensional table. The following rules apply for using the OCCURS clause to create multi-dimensional tables.

- To define a two-dimensional table, an OCCURS clause must appear in the data description of the element of the table and in the description of only one group item that contains that table element.
- To define a three-dimensional table, the OCCURS clause should appear in the data description of the element of the table and in the description of two group items that contain the element.

COBOL permits tables of up to 48 dimensions. Example 5–3 shows a table that has one dimension for CONTINENT-NAME, two dimensions for COUNTRY-NAME, and three dimensions for CITY-NAME and CITY-POPULATION. The table includes 100,510 data items: 10 for CONTINENT-NAME, 500 for COUNTRY-NAME, 50,000 for CITY-NAME and 50,000 for CITY-POPULATION. Within the CENSUS-TABLE there are ten occurrences of CONTINENT-TABLE. Within each CONTINENT-TABLE there are 50 occurrences of COUNTRY-NAME and CITY-TABLE, and within each CITY-TABLE there are 100 occurrences of CITY-NAME and CITY-POPULATION.

```
01  CENSUS-TABLE.
    05  CONTINENT-TABLE                    OCCURS 10 TIMES.
        10  CONTINENT-NAME          PIC X(6).
        10  COUNTRY-TABLE                  OCCURS 50 TIMES.
            15  COUNTRY-NAME        PIC X(8).
            15  CITY-TABLE                  OCCURS 100 TIMES.
                20  CITY-NAME       PIC X(10).
                20  CITY-POPULATION PIC 9(12).
```

Example 5–3. Three-Dimensional Table

Table Handling

Whenever you refer to a table element, the reference must indicate which occurrence of the element is intended. For access to a one-dimensional table, the occurrence number of the desired element provides complete information. For access to tables of more than one dimension, you must supply an occurrence number for each dimension of the table accessed. In Example 5–3, a reference to the fourth CONTINENT-NAME would be complete, whereas a reference to the fourth COUNTRY-NAME would not. To refer to COUNTRY-NAME, which is an element of a two-dimensional table, refer to the fourth COUNTRY-NAME within the sixth CONTINENT-TABLE.

Any table that appears in the Working-Storage Section of the program can be coded to contain specific initial values for each item. You do so by first declaring all values on individual data items in a group and then redefining this group as a table. Example 5–4 shows this type of table initialization.

```
01  MONTH-TABLE.
    05  THE-MONTHS.
        10  MON-1      PIC X(3)      VALUE "JAN".
        10  MON-2      PIC X(3)      VALUE "FEB".
        10  MON-3      PIC X(3)      VALUE "MAR".
        10  MON-4      PIC X(3)      VALUE "APR".
        10  MON-5      PIC X(3)      VALUE "MAY".
        10  MON-6      PIC X(3)      VALUE "JUN".
        10  MON-7      PIC X(3)      VALUE "JUL".
        10  MON-8      PIC X(3)      VALUE "AUG".
        10  MON-9      PIC X(3)      VALUE "SEP".
        10  MON-10     PIC X(3)      VALUE "OCT".
        10  MON-11     PIC X(3)      VALUE "NOV".
        10  MON-12     PIC X(3)      VALUE "DEC".
    05  MONTHS-REDEFINED REDEFINES THE-MONTHS.
        10  MONTH-ITEM  PIC X(3)    OCCURS 12.
```

Example 5–4. Table Initialization

Subscripting

One method used to specify a particular element of a multi-dimensional table is to append one or more subscripts to each table element data name. A subscript is an integer whose value specifies the occurrence number of an element. The subscript can be represented by either a data name or a numeric literal that is an integer. The data name must be a numeric elementary item that represents an integer. When the subscript is represented by a data name, the data name can be qualified, but not subscripted.

The subscript, enclosed in parentheses, is written immediately following the name of the table element. The table element data name appended with a subscript is called a subscripted data name or an identifier. When more than one subscript is required, the subscripts are written in order of successively less inclusive dimensions of the data organization.

The subscript can be signed. If it is signed, it must be positive. The lowest possible subscript value is 1. This value points to the first element of the table. The next sequential elements of the table are pointed to by subscripts whose values are 2, 3, and so on. The highest permissible subscript value, is the maximum number of occurrences of the item, as specified in the OCCURS clause.

An attempt to use a subscript whose value is outside this permissible range can terminate the program. Refer to "Subscript and Index Validation" in this section.

A table reference must include the data name being referenced and as many subscripts as there are dimensions in the preceding referenced table. That is, there must be a subscript for each OCCURS clause from the start of the table up to and including the statement that contains the desired data name. For example, in Example 5–3, references to CONTINENT-NAME require only one subscript, references to COUNTRY-NAME require two, and references to CITY-NAME and CITY-POPULATION require three.

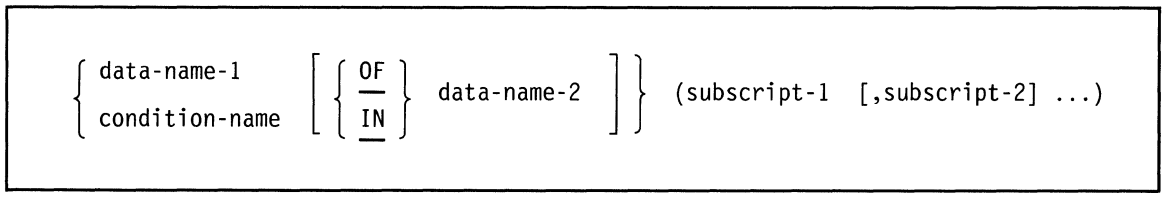
When a data name is used as a subscript, it can be used to refer to items in many different tables. These tables need not have elements of the same size. The data name can also appear as the only subscript with one item and as one of two or three subscripts with another item. Also, mixing literal and data name subscripts is permissible, as in the following example:

```
CITY-POPULATION(10, NEWKEY, 42)
```

In this example, the literal 10 refers to the occurrence number in the CONTINENT-TABLE. The data-name NEWKEY must be defined as an integer and represents the occurrence number in the COUNTRY-TABLE, and the literal 42 represents the occurrence number in the CITY-TABLE.

When you qualify a subscripted data name, the qualification must precede the subscript.

The following is the general format for subscripting:



Indexing

Another method of referring to items in a table is indexing. To index, you assign one or more index names to an item whose data description contains an OCCURS clause. You assign an index to that level of the table by using the INDEXED BY phrase in the table definition. A name given in the INDEXED BY phrase is known as an index name and is used to refer to the assigned index. The value of an index corresponds to the occurrence number of an element in the associated table.

Indexing may be direct or relative. Specify direct indexing by using an index name in the form of a subscript. Specify relative indexing by following the index name with either the plus (+) or the minus (−) operator followed by an unsigned integer numeric literal, all of which you enclose in parentheses and immediately follow with the table element data-name. The occurrence number resulting from relative indexing is determined by incrementing (when the operator + is used) or decrementing (when the operator − is used) the occurrence number represented by the value of the index, by the value of the literal.

Note: *The use of the + or − integer is permitted only with indexes, not with subscripts.*

An index name must be initialized before it is used as a table reference. An index name can be given a value by the execution of a SET statement, a SEARCH ALL statement, or a Format 4 PERFORM statement.

Just as with subscripts, when more than one index name is required, the names are written in the hierarchical order of successively fewer inclusive dimensions of the data organization. When a reference requires more than one occurrence number for completeness, you must not use a data name subscript to indicate one occurrence number and an index name for another. Therefore, if you use indexing, each OCCURS clause within the hierarchy (each dimension of the table) must contain an INDEXED BY clause. You can, however, mix literals and index names within one reference, just as you can mix literals and data name subscripts.

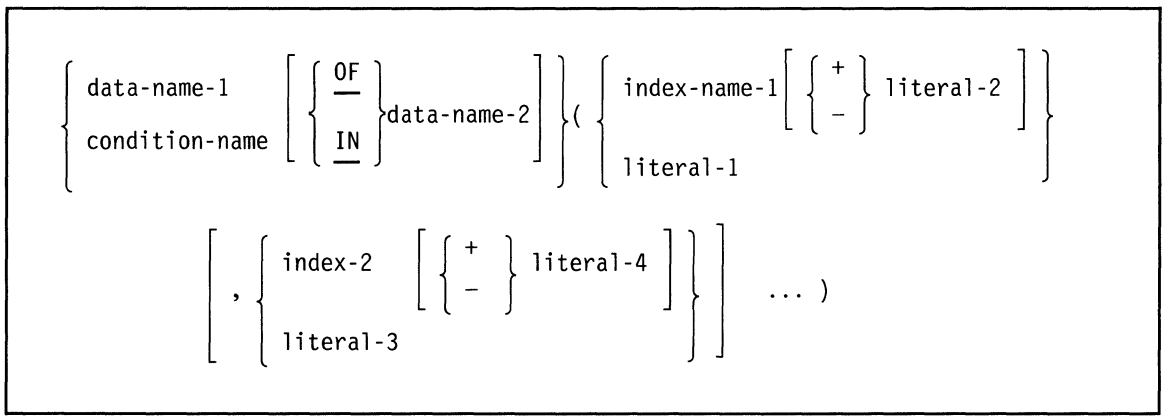
An index data item is an elementary data item whose data description contains a USAGE IS INDEX clause. An index-name assigned to one table cannot be used to index another table.

An index name has the same internal representation as an index data item. If a value to be stored in an index name or in an index data name exceeds the largest value that can be held in that index name or index data name, then the value is truncated according to the rules for a size error condition in an arithmetic statement without a SIZE ERROR phrase. Refer to "SIZE ERROR Phrase" in Section 6.

Subscripting is permitted wherever indexing is permitted.

When an indexed data name is qualified, the qualification must precede the index.

The following is the general format for indexing:



Subscript and Index Validation

The COBOL compiler generates code to do bounds checking at execution time for programs that use subscripting and/or indexing. This code changes program size and performance. If bounds checking is not desired, the dollar option NOBOUNDS is available. It must be specified at compile time to inhibit the generation of the bounds checking code.

When a statement referring to a subscripted or an indexed table element is executed, the value of each subscript or direct or relative index is validated. The value of each subscript must not be less than one or more than the maximum number of occurrences as specified by the corresponding OCCURS clause (as modified by any DEPENDING ON clause). The value of each index must not be less than a value that corresponds to the beginning of the last occurrence of the table element (as determined by the OCCURS clause).

Note: *The index value need not precisely address the beginning of a table element in order to pass the range check. This can occur when an index name is set to the value of an index data item that has been set to the value of another index name, because such assignments are made without conversion.*

If the subscript or index is not within the given range, the program terminates abnormally. A display message appears, giving the source record number of the statement with the error.

Indexing Versus Subscripting

Generally, indexing is superior to subscripting, since access to an item via an index can be more direct than access via a subscript.

The following definitions are necessary at this point:

- The operands in the INDEXED BY portion of an OCCURS clause are index names.
- The data item whose USAGE IS INDEX is an index data item.

Subscripts can be used to access tables for which index names are specified, but indexing cannot be used with tables not having index names.

The primary difference between subscripting and indexing is in the method used to access the desired table entry.

At object time, a subscript is an integer representing an occurrence within a table, that is, 1 for the first entry, 2 for the second, and so forth.

Since the subscript data item contains only an occurrence number of the item to be accessed, the program must multiply the occurrence number by the length of a table entry in order to locate the desired item.

For example, assume a table defined as Y PICTURE 9(4) COMP OCCURS 10 TIMES INDEXED BY NDX. Assume that a numeric data item SUB has also been declared for use as a subscript.

In order to execute the statement MOVE Y(SUB) TO X, the program must first multiply the value in SUB by 10, the length of the table element Y. Doing so gives the offset of the desired element from the beginning of the table. Then the

location of the item can be found by adding the offset, less the length of one table element, to the beginning address of the table.

Each time SUB is used to access an item, the multiply operation is repeated.

Literal subscripts (for example, Y(5)) are calculated once at compile time and therefore involve the same code as nonsubscripted items.

An index name or index data item contains an offset into a table, rather than a simple occurrence number. The statement *SET NDX TO data name* causes the program to compute the offset of the element whose position is given by data name. The statements SET NDX UP BY 1 and SET NDX DOWN BY 1 cause the length of one table item to be added to, or subtracted from, the value in NDX.

In the statement MOVE Y(NDX) TO X, the location of Y is determined by adding the contents of NDX, less the length of one table element, to the beginning address of the table. The multiply operation required for the subscript is not required for the index.

The index value is always carried in its final form, as an offset. The multiplication is performed, when applicable, at the time the index is set, not each time it is used, as is the case with subscripting.

Another advantage of indexing is that relative indexing is permitted. For example, the statement MOVE Y(NDX+2) TO X accesses the second table element beyond the one currently pointed to by NDX; the statement ADD 3 TO Y(NDX−1) accesses the element preceding the one pointed to by NDX. Such relative accessing is not possible with subscripting.

Relative indexing does not change the value in the index name.

The SEARCH verb can only be used with a table defined with an INDEXED BY clause.

Index data items are intended only for the temporary storage of the contents of specified index names. They can be used only in SEARCH and SET statements and in relational conditions; they cannot be used to access table entries.

Indexing can lose its advantage over subscripting with excessive use of index names for purposes other than indexing, such as in the statement *IF NDX IS GREATER THAN data name.*

It is more efficient to compare an index name to an index data name than to a non-index data name. When an index must be compared to a literal value (for example, to the number of entries in a table, to determine when to end a loop), the index name should be set to the literal value in the program's housekeeping area. An index data item is then set to the index name, and that index data item is then used in all comparison operations where the literal value would otherwise be used.

Refer to "OCCURS Clause" in this section, and to "SET Statement" and "SEARCH Statement" in Section 6 for further information.

File Section

The File Section describes the structure of data files. Each file is defined by a file description entry and one or more record description entries. Record descriptions are written immediately following the file description entry.

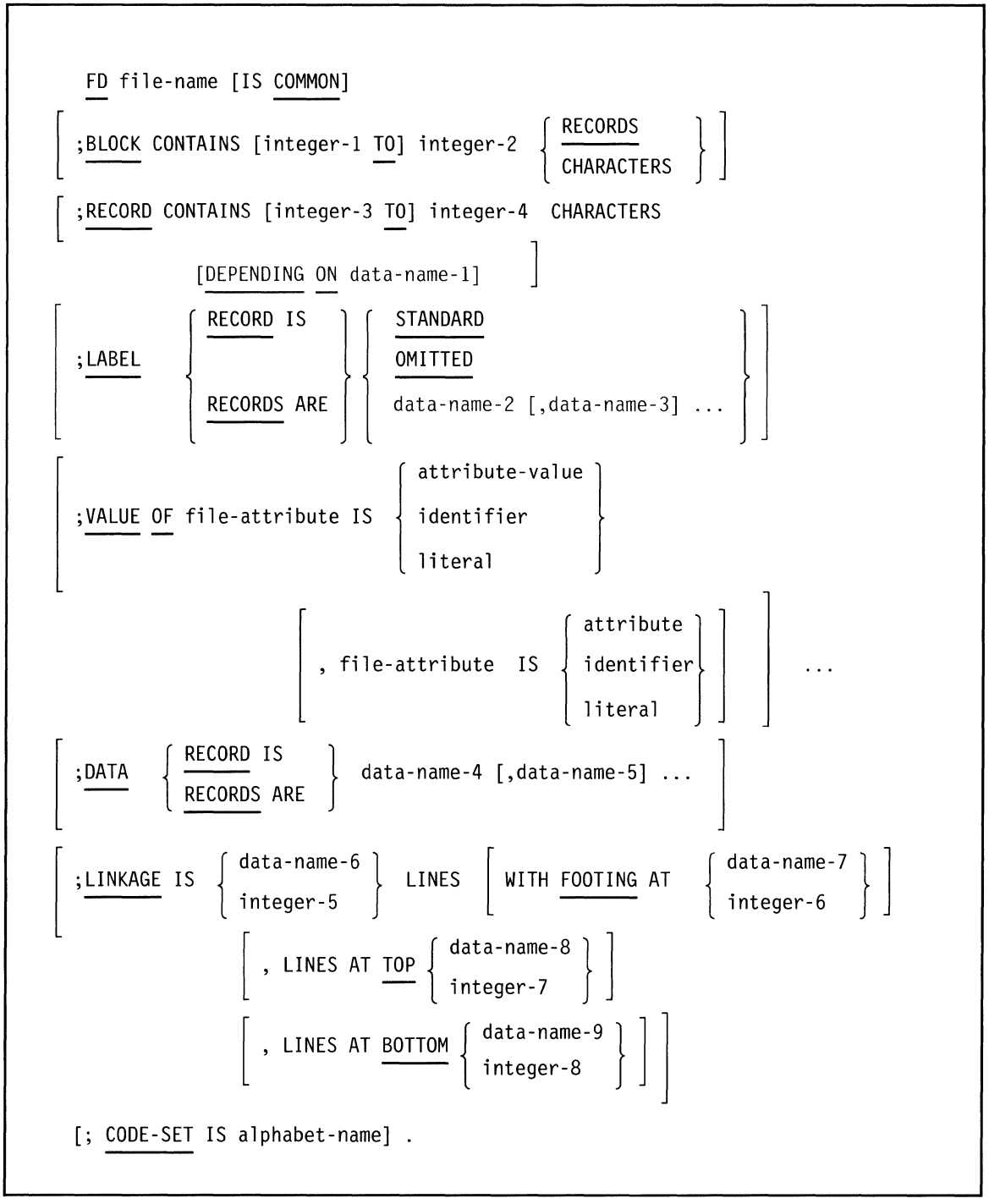
File Description Entry

The file description furnishes information concerning the physical structure, identification, and record names pertaining to a given file (see Format 1).

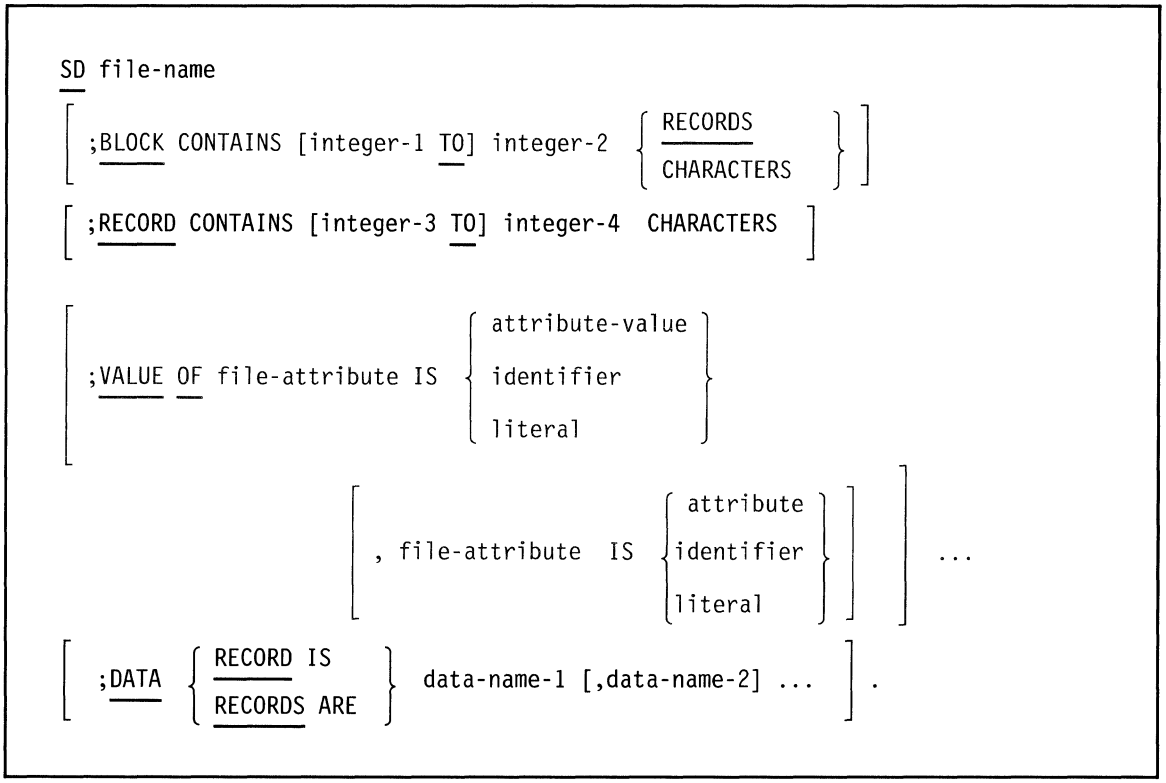
The sort-merge file description furnishes information concerning the physical structure, identification, and record names of the file to be sorted or merged (see Format 2). The items in the file description entry can appear in any order

The following are the general formats:

Format 1


FD file-name [IS COMMON]
;BLOCK CONTAINS [integer-1 TO] integer-2 { RECORDS / CHARACTERS }
;RECORD CONTAINS [integer-3 TO] integer-4 CHARACTERS
[DEPENDING ON data-name-1]
;LABEL { RECORD IS / RECORDS ARE } { STANDARD / OMITTED / data-name-2 [,data-name-3] ... }
;VALUE OF file-attribute IS { attribute-value / identifier / literal }
, file-attribute IS { attribute / identifier / literal } ...
;DATA { RECORD IS / RECORDS ARE } data-name-4 [,data-name-5] ...
;LINKAGE IS { data-name-6 / integer-5 } LINES [WITH FOOTING AT { data-name-7 / integer-6 }]
, LINES AT TOP { data-name-8 / integer-7 }
, LINES AT BOTTOM { data-name-9 / integer-8 }
[; CODE-SET IS alphabet-name] .

Format 2


SD file-name
;BLOCK CONTAINS [integer-1 TO] integer-2 { RECORDS / CHARACTERS }
;RECORD CONTAINS [integer-3 TO] integer-4 CHARACTERS
;VALUE OF file-attribute IS { attribute-value / identifier / literal }
, file-attribute IS { attribute / identifier / literal }
...
;DATA { RECORD IS / RECORDS ARE } data-name-1 [,data-name-2] ... .

Level Indicators (FD,SD)

In a COBOL program, the file description entry (FD) represents the highest level of organization in the File Section. The File Section header is followed by a file description entry consisting of a level indicator FD, a file name, and a series of independent clauses. The FD clauses specify the size of the logical and physical records, the presence or absence of label records, the value of file-attributes, the names of the data records that comprise the file, and the number of lines to be written on a logical printer page. The entry itself is terminated by a period.

A sort-merge file description entry (SD) gives information about the size and the name of the data records associated with the file to be sorted or merged. There are no label procedures that the user can control, and the rules for blocking and internal storage are peculiar to the SORT and MERGE statements.

The level indicator FD identifies the beginning of a file description and must precede the file name.

The level indicator SD identifies the beginning of the sort-merge file description and must precede the file name.

One or more record description entries must follow the file description entry. One or more record description entries must also follow the sort-merge file description entry; however, no input/output statements can be executed for this file.

BLOCK CONTAINS Clause

The BLOCK CONTAINS clause specifies the size of a physical record.

The following is the general format:

```
BLOCK CONTAINS   [integer-1 TO ]   integer-2  { RECORDS    }
                                              { CHARACTERS }
```

This clause is required when the physical record (a block) contains more than one logical record. If this clause is not specified, the physical record is presumed to contain only one logical record that is as large as the largest record specified for this file. Such a file is referred to as unblocked.

This clause can be specified for a sort-merge file. For sort-merge files, the maximum block size is 19998 characters.

When the word RECORDS is specified, the physical record size is considered to be integer-2 times the largest record specified for this file.

When the word CHARACTERS is specified, the physical record size is considered to be integer-2 characters. If integer-2 is not a multiple of the largest record specified for this file, the physical record size is adjusted so that it is a multiple of the largest record specified, but not to exceed the value of integer-2. Integer-2 must not be less than the largest record specified for the file.

If only integer-2 is shown, it represents the exact number of RECORDS or CHARACTERS in the physical record unless the size is adjusted as stated when CHARACTERS is specified.

If integer-1 and integer-2 are both shown, they refer to the minimum and maximum size of the physical record, respectively.

If logical records of differing sizes are grouped into one physical record, the amount of data transferred from the record area to the physical record depends on the size of the record named in the WRITE or REWRITE statement. The logical records are aligned on maximum-record-size boundaries. If the size of the record named is not equal to the maximum-record-size specified for the file, the data is transferred to the physical record according to the rules specified for the MOVE statement without the CORRESPONDING phrase. For this MOVE, the sending area is treated as a group item.

If variable length records are specified (see "RECORD CONTAINS clause" in this section), then the block size is determined as follows:

- If the word RECORDS is shown and only integer-2 is shown, the block size is equal to integer-2 times the maximum-record-size.
- If the word RECORDS is shown with integer-1 and integer-2 both shown, the block size is equal to the larger of integer-1 times the maximum-record-size and integer-2 times the maximum-record-size. The maximum number of records allowed is 999.
- If the word CHARACTERS is shown, the block size is equal to the larger of integer-2 and maximum-record-size. If maximum-record-size is larger, a warning is issued. If integer-1 is shown, it is used for documentation only.

Note: *The compiler generates code within the object program to deblock most blocked files, with the exception of sequential files opened I/O.*

Variable length records can be declared for disk, diskpack, and tape files.

CODE-SET Clause

The CODE-SET clause specifies the character code set used to represent data on the external media.

The following is the general format:

```
CODE-SET IS alphabet-name
```

When the CODE-SET clause is specified for a file, all data in that file must be described as USAGE IS DISPLAY, and any signed numeric data must be described with the SIGN IS SEPARATE clause.

You can specify the CODE-SET clause only for non- mass storage files.

If you specify the CODE-SET clause, use the alphabet-name to designate the character code convention to represent data on the external media. The alphabet-name also specifies the algorithm for converting the character codes on the external media from or to the native character codes. This code conversion occurs when an input or output operation is executed (see "SPECIAL-NAMES Paragraph" in Section 4).

Alphabet-name can be either EBCDIC or ASCII.

If the CODE-SET clause is not specified, the EBCDIC code set is assumed for data on the external media.

COMMON Clause

The COMMON clause specifies that a file is common to, or shared with, one or more programs.

The following is the general format:

```
IS COMMON
```

When used in an FD entry, the COMMON clause indicates that the file and its record description entries are shared among programs.

Note: *PORT files can be shared between COBOL and Pascal programs. Files other than indexed and relative files can only be shared among COBOL programs.*

For further information, refer to "CALL MODULE Statement" in Section 6 and to Section 10, "COBOL Binding".

DATA RECORDS Clause

The DATA RECORDS clause serves only as documentation for the names of data records with their associated file.

The following is the general format:

```
DATA { RECORD IS  } data-name-1 [ , data-name-2 ] ...
     { RECORDS ARE }
```

Data-name-1 and data-name-2 are the names of data records for this file. Each must be defined by a level number 01 record description entry of the same name within this FD.

The presence of more than one data name indicates that the file contains more than one type of data record. These records can be of differing sizes, different formats, and so on. The order in which they are listed is not significant.

Conceptually, all data records within a file share the same area. This concept is in no way altered by the presence of more than one type of data record within the file.

Example 5–5 shows coding of an FD and a DATA RECORDS clause.

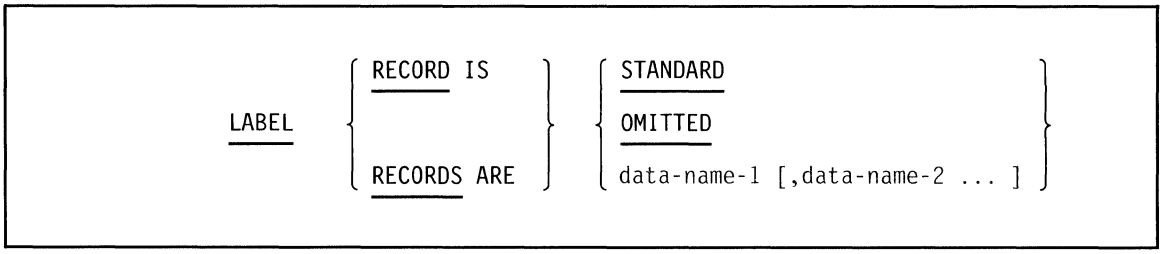
```
FD  INPUT-FILE
    DATA RECORDS ARE PRODUCTION, SALES, INVENTORY.
 01 PRODUCTION.
    03  REC-TYPE-1      PIC 99.
    03  REC-PROD        PIC X(78).
 01 SALES.
    03 REC-TYPE-2       PIC 99.
    03 REC-SALES        PIC X(78).
 01 INVENTORY.
    03 REC-TYPE-3       PIC 99.
    03 REC-INVEN        PIC X(78).
```

Example 5–5. Coding FD and DATA RECORDS

LABEL RECORDS Clause

The LABEL RECORDS clause specifies whether labels are present.

The following is the general format:



This clause is not required. If it is not specified, the STANDARD option is assumed.

The LABEL RECORDS clause is ignored for mass storage files. All mass storage files have standard labels.

STANDARD specifies that labels exist for the file or the device to which the file is assigned, and the labels conform to the Unisys label specifications.

OMITTED specifies that no explicit labels exist for the file or the device to which the file is assigned.

Data-name-1 and data-name-2 identify the label descriptions to be used by the USE procedures for a given file. They can be part of the data description of the file or they can be in Working-Storage. The subordinate items define the label area and identify the fields to be accessed in the label records. The label definition takes precedence over a VALUE OF LABELSIZE clause if both are specified and the label definition is larger. The VALUE OF LABELKIND clause is used to identify the type of labels used.

The STANDARD label record serves as both the beginning and ending label records. It is described in Table 5–2. The label fields that cannot be changed are indicated in the middle column. Before the label is physically written, the MCP overwrites anything put into these fields by the user program.

The MCP overwrites the first 80 characters of the MSUSASII label. An example of a program that accesses tape labels is included in Appendix E.

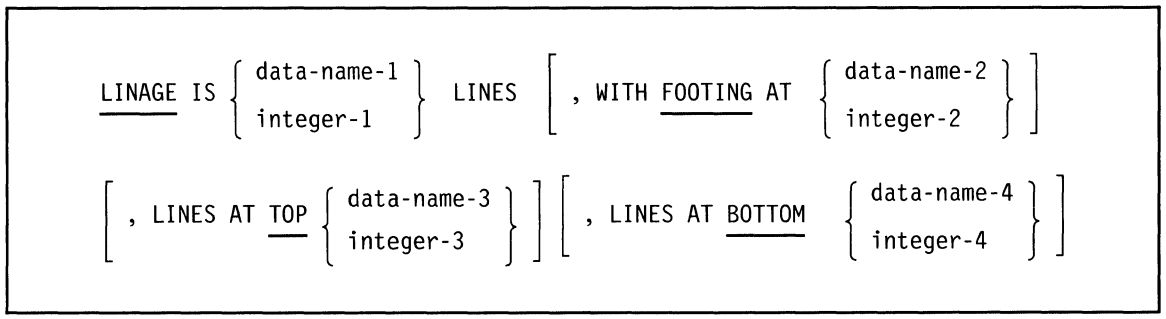
Table 5–2 . STANDARD Label Record Description

Positions	Can Be Changed	Field Description
1–8	No	" LABEL "
9	Yes	Zero
10–15	Yes	Multifile-id, or zeros
16	Yes	Blank
17	Yes	Zero
18–23	Yes	File identifier
24	Yes	Blank
25–27	No	Reel number
28–32	No	Creation date (YYDDD)
33–34	Yes	Cycle (default 01)
35–39	No	Purge date (YYDDD)
40	No	Sentinel (ending label only) 0 = End of file 1 = End of reel
41–45	No	Block count (ending label only)
46–52	No	Record count (ending label only)
53	Yes	Reserved field
54–58	No	Physical tape number
59–80	Yes	Reserved field
81+	Yes	User defined field

LINAGE Clause

The LINAGE clause provides a means for specifying the depth of a logical page in terms of number of lines. This clause also provides for specifying the size of the top and bottom margins on the logical page, and the line number, within the page body, at which the footing begins.

The following is the general format:


LINAGE IS { data-name-1 | integer-1 } LINES [, WITH FOOTING AT { data-name-2 | integer-2 }]
[, LINES AT TOP { data-name-3 | integer-3 }] [, LINES AT BOTTOM { data-name-4 | integer-4 }]

Data-name-1, data-name-2, data-name-3, data-name-4 must reference elementary unsigned numeric integer data items.

The value of integer-1 must be greater than 0.

The value of integer-2 must not be greater than integer-1.

The value of integer-3 and integer-4 can be 0.

The LINAGE clause provides a means for specifying the size of a logical page in terms of number of lines. The logical page size is the sum of the values referenced by each phrase except the FOOTING phrase. If the LINES AT TOP or LINES AT BOTTOM phrases are not specified, the values for these functions are 0. If you do not specify the FOOTING phrase, the assumed value is equal to either integer-1 or the contents of the data item referenced by data-name-1, whichever is specified.

There is not necessarily any relationship between the size of the logical page and the size of the physical page.

The value of integer-1 or the data item referenced by data-name-1 specifies the number of lines that can be written and/or spaced on the logical page. The value must be greater than 0. That part of the logical page in which these lines can be written and/or spaced is called the page body.

The value of integer-3 or the data item referenced by data-name-3 specifies the number of lines that comprise the top margin on the logical page. The value can be 0.

The value of integer-4 or the data item referenced by data-name-4 specifies the number of lines that comprise the bottom margin on the logical page. The value can be 0.

The value of integer-2 or the data item referenced by data-name-2 specifies the line number within the page body at which the footing area begins. The value must be greater than 0 but not greater than either the value of integer-1 or the data item referenced by data-name-1.

The footing area comprises the area of the logical page between the line represented by the value of integer-2 or the data item referenced by data-name-2 and the line represented by the value of integer-1 or the data item referenced by data-name-1, inclusive.

The value of integer-1, integer-3, and integer-4, if specified, is used when you open the file by executing an OPEN statement with the OUTPUT phrase to specify the number of lines that comprise each of the indicated sections of a logical page. The value of integer-2, if specified, is used at that time to define the footing area. These values are used for all logical pages during an execution of the program.

The values of the data items referenced by data-name-1, data-name-3, and data-name-4, if specified, are used as follows:

- The values of the data items at the time an OPEN statement with the OUTPUT phrase is executed for the file, is used to specify the number of lines that are to comprise each of the indicated sections for the first logical page.
- At the time a WRITE statement with the ADVANCING PAGE phrase is executed or page overflow condition occurs (see "WRITE Statement" in Section 6), the values of the data items are used to specify the number of lines that are to comprise each of the indicated sections for the next logical page.

The value of the data item referenced by data-name-2, if specified at the time an OPEN statement with the OUTPUT phrase is executed for the file, is used to define the footing area for the first logical page. At the time a WRITE statement with the ADVANCING PAGE phrase is executed or a page overflow condition occurs, it is used to define the footing area for the next logical page.

A LINAGE-COUNTER is generated by the presence of a LINAGE clause. The value in the LINAGE-COUNTER at any given time represents the line number at which the device is positioned within the current page body.

Rules of Use

The rules governing the LINAGE-COUNTER are as follows:

- A separate LINAGE-COUNTER is supplied for each file described in the File Section whose file description entry contains a LINAGE clause.
- LINAGE-COUNTER can be referenced, but cannot be modified, by Procedure

Division statements. Since more than one LINAGE-COUNTER can exist in a program, the user must qualify LINAGE-COUNTER by file-name when necessary.

- LINAGE-COUNTER is automatically modified according to the following rules during the execution of a WRITE statement to an associated file:
  - When the ADVANCING PAGE phrase of the WRITE statement is specified, the LINAGE-COUNTER is automatically reset to 1.
  - When the ADVANCING identifier-2 or integer phrase of the WRITE statement is specified, the LINAGE-COUNTER is incremented by the specified integer or the value of the data item referenced by identifier-2.
  - When the ADVANCING phrase of the WRITE statement is not specified, the LINAGE-COUNTER is incremented by 1 (see "WRITE Statement" in Section 6).
  - The value of LINAGE-COUNTER is automatically reset to 1 when the device is repositioned to the first line that can be written on for each of the succeeding logical pages (see "WRITE Statement").

The value of LINAGE-COUNTER is automatically set to 1 at the time an OPEN statement is executed for the associated file.

Each logical page is contiguous with the next, with no additional spacing provided.

Practical Considerations

When a LINAGE clause is specified for a file, the compiler generates code to count the number of lines written to that file.

When a file description includes a LINAGE clause specifying PAGE in a WRITE statement for that file, an advance to the beginning of the next logical page results. You do so by writing a sufficient number of lines to position the file at the beginning of the logical page.

Note that in the case of a printer file, this means that a CHANNEL 1 indicator is ignored when a LINAGE clause is present; page advance occurs solely by writing lines. Doing so has two effects:

- Printer backup files created with LINAGE might be larger than those created without it, since a page skip can generate multiple individual writes instead of a single skip to a channel.
- The size of a logical page has no relation to the size of a physical page except as the programmer specifies it.

For example, if a report is to be printed on forms having 11-inch page lengths and the printer spacing is to be 6 lines per inch, the LINAGE clause must account for all 66 available lines if printed information is to appear where expected. That is, the sum of the integers in the LINAGE, LINES AT TOP, and LINES AT BOTTOM phrases must be 66.

RECORD CONTAINS Clause

The RECORD CONTAINS clause specifies the size of the data records.

The following is the general format:

```
RECORD CONTAINS  [integer-1 TO]  integer-2 CHARACTERS

          [ DEPENDING ON data-name-1 ]
```

The size of each data record is completely defined within the record description entry; therefore, this clause is never required. When present, however, the following notes apply:

- At compile time, the size (as specified by this clause) is compared against the actual size of the record description entry for consistency.
- Integer-2 cannot be used by itself unless all the data records in the file are the same size. In this case, integer-2 represents the exact number of characters in the data record. If integer-1 and integer-2 are both shown, they refer to the minimum number of characters in the smallest data record and the maximum number of characters in the largest data record, respectively.
- The size is specified in terms of the number of character positions required to store the logical record regardless of the types of characters used to represent the items within the logical record. The size of a record is determined by the sum of the number of characters in all fixed length elementary items plus the sum of the maximum number of characters in any variable length item subordinate to the record. This sum can be different from the actual size of the record.

If integer-1 and integer-2 are both shown, it is implied that the logical records are variable in length, in which case the following notes apply:

- If the DEPENDING clause is not specified, the logical record length is automatically derived by the compiler and supplied as the first four characters of the record when it is written. This record length portion of the record cannot be referenced.
- If the DEPENDING clause is specified, the logical record length is supplied by the programmer in data-name-1 at the time the record is written, and the following notes apply:
  - Data-name-1 must occupy the first four characters of each record of the file.
  - The logical record length includes the four characters needed to contain the length.

Data-name-1 must reference an elementary unsigned numeric integer four characters in length.

When variable length records are specified, the block size is determined by the BLOCK CONTAINS clause (see "BLOCK CONTAINS Clause" earlier in this section).

Variable length records can be declared for disk, diskpack, and tape files.

For sort-merge files the maximum record size is 4860 characters for diskpack files, 4800 characters for disk files, and 4998 characters for tape files.

VALUE OF Clause

The VALUE OF clause can be used to assign values to certain file attributes, that is, items within the File Information Block (FIB) and to label areas associated with a file.

This clause can be used in a sort-merge file description entry.

The following is the general format:

```
                                    { attribute-value }
VALUE OF file-attribute IS          { identifier      }
                                    { literal         }

   [                                { attribute-value } ]
   [ ,file-attribute IS             { identifier      } ]  ...
   [                                { literal         } ]
```

Example 5–6 depicts coding of this VALUE clause. Section 7 contains a complete discussion and a list of all file attributes and the appropriate values for each attribute. Only those attributes that can be written (an RW or WO in column 4, Read/Write, of Table 7–1) can be used in the VALUE OF clause.

```
*
*REFER TO EXAMPLE 5-1 FOR THE FILE-CONTROL ENTRIES FOR THE
*    FOLLOWING FDS.
*
 DATA DIVISION.
 FILE SECTION.
 FD  INPUT-2
     LABEL RECORDS ARE STANDARD
     VALUE OF FILENAME "TAPEIN"
              SAVEFACTOR 90.
 01  TAPE-REC  PIC X(80).
 FD  WORK-FILE
     VALUE OF FAMILYNAME "PACK01"
              AREAS 20
              AREASIZE 1000
              FILENAME "TEMP01".
 01  WORK-REC.
     05  KEY-REC       PIC 9(8) COMP.
     05  REM-WORK      PIC X(76).
```

Example 5–6. Coding the VALUE OF Clause

Record Description

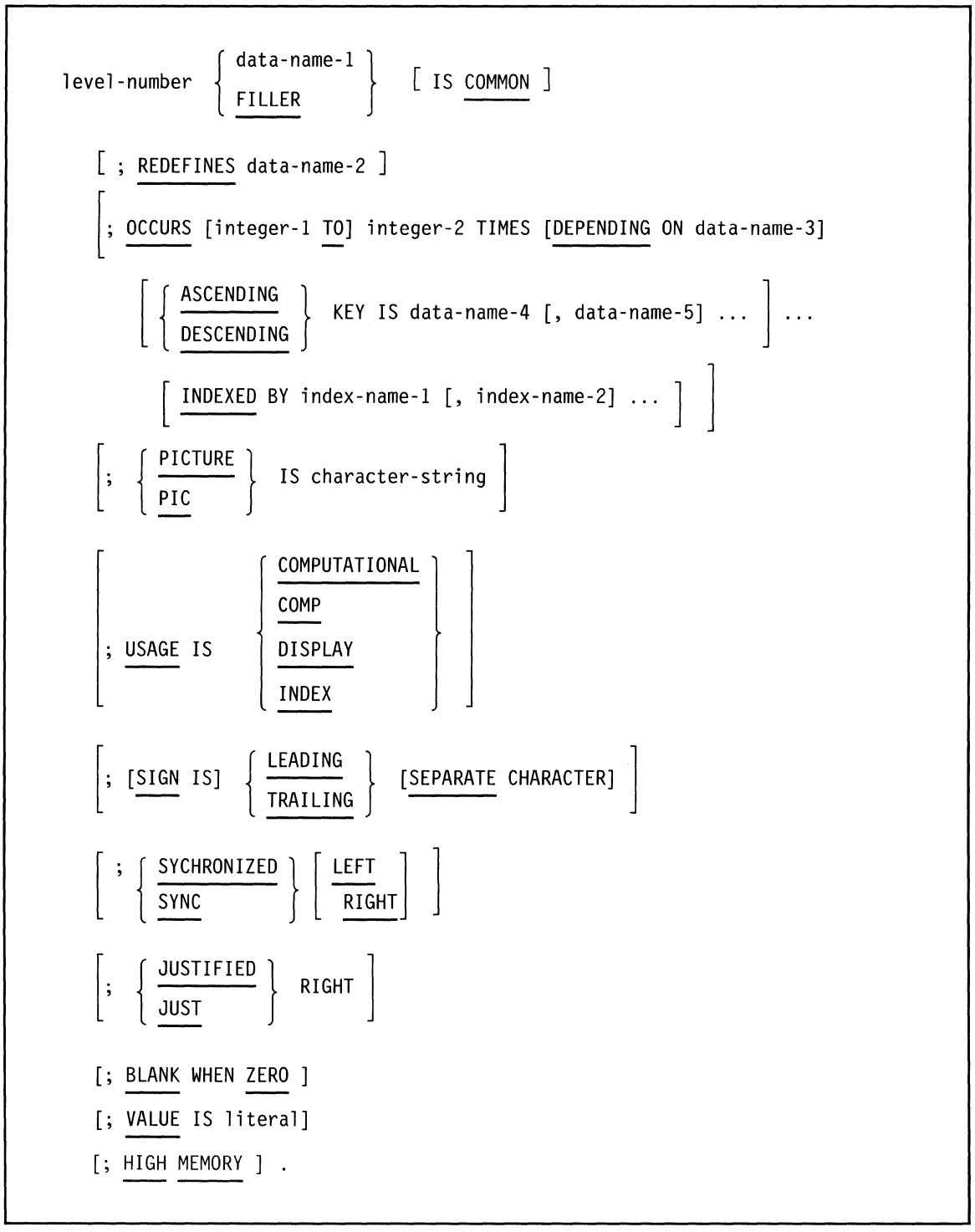
A record description consists of a set of data description entries that describe the characteristics of a particular record. Each data description entry consists of a level number followed first by a data name, if required, and then by a series of independent clauses, as required. A record description has a hierarchical structure, and, therefore, the clauses used with an entry can vary considerably, depending upon whether or not the entry is followed by subordinate entries.

Data Description Entry

A data description entry specifies the characteristics of a particular item of data.

The following are the general formats:

Format 1


level-number { data-name-1 / FILLER } [IS COMMON]
[; REDEFINES data-name-2]
[; OCCURS [integer-1 TO] integer-2 TIMES [DEPENDING ON data-name-3]
[{ ASCENDING / DESCENDING } KEY IS data-name-4 [, data-name-5] ...] ...
[INDEXED BY index-name-1 [, index-name-2] ...]]
[; { PICTURE / PIC } IS character-string]
[; USAGE IS { COMPUTATIONAL / COMP / DISPLAY / INDEX }]
[; [SIGN IS] { LEADING / TRAILING } [SEPARATE CHARACTER]]
[; { SYCHRONIZED / SYNC } [LEFT / RIGHT]]
[; { JUSTIFIED / JUST } RIGHT]
[; BLANK WHEN ZERO]
[; VALUE IS literal]
[; HIGH MEMORY] .

Format 2

```
66 data-name-1; RENAMES data-name-2 [ { THROUGH } data-name-3 ].
                                      { THRU    }
```

Format 3

```
88 condition-name; { VALUE IS  } literal-1 [ { THROUGH } literal-2 ]
                   { VALUES ARE }           { THRU    }

   [ , literal-3 [ { THROUGH } literal-4 ] ] ...
                   { THRU    }
```

The clauses can be written in any order, with two exceptions: the data-name-1 or FILLER clause must immediately follow the level number, and the REDEFINES clause, when used, must immediately follow the data-name-1 clause.

The level number in Format 1 can be any number from 01 through 49 or 77.

The PICTURE clause must be specified for every elementary item except an index data item, in which case use of this clause is prohibited.

The clauses SYNCHRONIZED, PICTURE, JUSTIFIED, and BLANK WHEN ZERO must not be specified except for an elementary data item.

Each record of a file begins with the level number 01 (which can also be shown as 1). This number is reserved for the record-name only, as the most inclusive grouping for a record. Less-inclusive groupings are given higher numbers, but not necessarily successive ones. The numbers can range up to 49. Example 5–7 illustrates the use of levels within a record.

The smallest element of a data description is called an elementary item. An elementary item cannot have subordinate levels. In Example 5–7, MONTH, DAYS, YEAR, MILLING, and FINISHING are elementary items. Since ITEM-NO, LOT-NO, STANDARD-COST, ASSEMBLY, INSPECTION, and WARRANTY-CODE do not have subsidiary clauses, they also represent elementary items.

A level that has further subdivisions is called a group item. In Example 5–7, ITEM-DATE, PRODUCTION-CODE, and MACHINE-SHOP represent items on a group level. A group is defined as being composed of all group and elementary items described under it. A group item ends when a level number is encountered of equal or lower numeric value than the group item itself. In Example 5–7, group item PRODUCTION-CODE ends with INSPECTION. A group item can only consist of a level number and a data-name, optionally followed by a VALUE, USAGE, or REDEFINES clause, followed by a period. COBOL defines all group items as alphanumeric.

All group items are byte-aligned by the compiler. All record (level number 01) items are word-aligned (one word is two bytes). The FILLER ADDED message appears where such alignment has taken place. Apart from level numbers 01 through 49, three additional level numbers exist in COBOL: 66, 77, and 88. It is important to remember that multiple level 01 entries of a given File Description in the File Section represent redefinition of the same memory area.

```
01  PRODUCTION-RECORD.                    *Record name
    03  ITEM-NO              PIC 9(5).    *Elementary Item
    03  LOT-NO               PIC 9(6).    *Elementary Item
    03  ITEM-DATE.                        *Group Item
        05  MONTH            PIC 99.      *Elementary Item
        05  DAYS             PIC 99.      *Elementary Item
        05  YEAR             PIC 99.      *Elementary Item
    03  STANDARD-COST        PIC 9(5)V99. *Elementary Item
    03  PRODUCTION-CODE.                  *Group Item
        05  MACHINE-SHOP.                 *Group Item
            07  MILLING      PIC 999.     *Elementary Item
            07  FINISHING    PIC 99.      *Elementary Item
        05  ASSEMBLY         PIC 9(4).    *Elementary Item
        05  INSPECTION       PIC X(5).    *Elementary Item
    03  WARRANTY-CODE        PIC XX.      *Elementary Item
```

Example 5–7. Elementary and Group Items

Clauses in the Data Description Entry

On the pages that follow are descriptions and formats for clauses that can occur in a data description.

BLANK WHEN ZERO Clause

The BLANK WHEN ZERO clause permits the blanking of an item when its value is 0.

The following is the general format:

```
BLANK WHEN ZERO
```

The BLANK WHEN ZERO clause can be used only for an elementary item whose PICTURE is specified as numeric or numeric-edited (see "PICTURE clause" in this section).

When the BLANK WHEN ZERO clause is used, the item contains nothing but spaces if the value of the item is 0.

When the BLANK WHEN ZERO clause is used for an item whose PICTURE is numeric, the category of the item is considered to be numeric-edited.

COMMON Clause

The COMMON clause specifies that a record item is common to, or shared with, one or more programs.

The following is the general format:

```
IS COMMON
```

When the COMMON clause is used in a data description entry, the following rules apply:

- The COMMON clause can be specified only on a record level (level number 01) item.
- The COMMON clause must appear on any data item to be shared with another module. When the clause is not present, a data item is assumed to be local to the module in which it is declared. That is, it cannot be accessed by any other module.
- If two or more COBOL modules are to refer to the same data item, each must declare that item in its own Data Division. The data description entries must each include the COMMON clause. The name given to the record (level number 01) item and the descriptions of any subordinate items within the data descriptions must be identical in each program declaring the COMMON item. File Section 01 data items and Working-Storage Section 01 data items need only be equated to the respective 01 data items in the other COBOL modules.

For further information, refer to "CALL MODULE Statement" in Section 6 and to Section 10 "COBOL Binding".

Condition-Names

A condition-name is a name assigned to a specific value, set of values, or range of values within a complete set of values that a data-item can assume. The data-item itself is called a conditional variable.

The following is the general format:

```
88 condition-name; { VALUE IS   } literal-1 [ { THROUGH } literal-2 ]
                   { VALUES ARE }             { THRU    }

[ , literal-3 [ { THROUGH } literal-4 ] ] ...
              { THRU    }
```

Note: *The VALUE clause discussed later in this section is included here but has been placed in this format to complete and clarify it.*

Format 3 of the data description entry is used for each condition name. Each condition name requires a separate entry with level number 88. Format 3 contains the name of the condition and the value, values, or range of values associated with the condition name. The condition name entries for a particular conditional variable must follow the entry describing the item with which the condition name is affiliated. A condition name can be associated with any data description entry that contains a level number, except for the following:

- Another condition name
- A level 66 item
- A group item containing items for which USAGE other than USAGE IS DISPLAY is specified either explicitly or implicitly
- An index data item (see "USAGE Clause" in this section)

Example 5–8 shows an example of condition name coding. The THIS-YEAR clause is the conditional variable, and the months JANUARY through DECEMBER are the condition names. JUNE, for example, has the assigned value 6. Using the condition name JUNE, the programmer could write the following:

```
IF JUNE PERFORM...
```

Doing so is logically equivalent to using the conditional variable THIS-YEAR and writing

```
IF THIS-YEAR IS EQUAL TO 06 PERFORM...
```

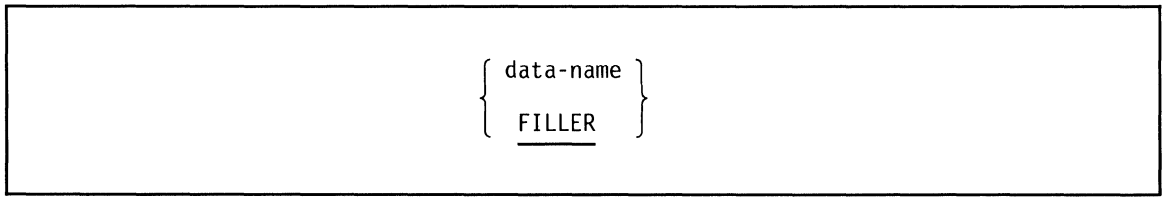
```
01  THIS-YEAR       PIC 99.
    88  JANUARY             VALUE 1.
    88  FEBRUARY            VALUE 2.
    88  MARCH               VALUE 3.
    88  APRIL               VALUE 4.
    88  MAY                 VALUE 5.
    88  JUNE                VALUE 6.
    88  JULY                VALUE 7.
    88  AUGUST              VALUE 8.
    88  SEPTEMBER           VALUE 9.
    88  OCTOBER             VALUE 10.
    88  NOVEMBER            VALUE 11.
    88  DECEMBER            VALUE 12.
    88  SPRING              VALUES 3 THRU 5.
    88  THIRTY-DAYS         VALUES 9, 4, 6, 11.
```

Example 5–8. Coding Condition Names

Data Name or FILLER Clause

A data name specifies the name of the data being described. The word FILLER specifies an elementary item of the logical record that cannot be referred to explicitly.

The following is the general format:



In the File and Working-Storage Sections, a data name or the keyword FILLER must be the first word following the level number in each data description entry.

The keyword FILLER can be used to name an elementary item in a record. Under no circumstances can a FILLER item be referred to explicitly. However, the keyword FILLER can be used as a conditional variable because such use does not require explicit reference to the FILLER item, but to its value.

HIGH MEMORY Clause

The HIGH MEMORY clause specifies that a data entry is to be allocated into the program's HIGH MEMORY areas.

The following is the general format:

```
HIGH MEMORY
```

You can specify the HIGH MEMORY clause only on a record level (level number 01) item and use it on any data item to be allocated at memory locations > 1 MD. It can appear in any order with other data description clauses.

All other data description clauses are still applicable in HIGH MEMORY areas. For example, a data item that redefines another data item located in HIGH MEMORY is also in HIGH MEMORY. Data items in HIGH MEMORY can be declared COMMON to other COBOL modules. In that case, each module must declare the data items to be in HIGH MEMORY. You can reference HIGH MEMORY data items in COBOL statements the same way as you do any other data items.

The MCP keyboard input command DC displays the program name, date compiled, compiler type, and memory requirements. The memory requirements returned for programs that use HIGH MEMORY are the initial program size (Memory Area 1). In addition, the $MAP compiler option can help you develop programs that access more than 1 MD of data. This improved MAP summary includes the following information:

- Details of HIGH MEMORY data areas
- Run time memory requirements
- Memory area layouts

You must take into account the following restrictions when using HIGH MEMORY:

- A HIGH MEMORY data area is limited to 1 MD in size.
- Data cannot be passed using IPC calls with parameters allocated in HIGH MEMORY.
- A HIGH MEMORY clause cannot be used in a program where $DMSICM is set.
- Data items that reference SDF forms cannot be allocated in HIGH MEMORY.
- Data entries declared with the HIGH MEMORY clause cannot be used either in the USING phrase of the interprogram communication CALL statement or in the Procedure Division header.
- Data items in HIGH MEMORY can be declared COMMON only to other COBOL modules. These data items cannot be referenced by BPL, FORTRAN ANSI-77 or Pascal ICMs.

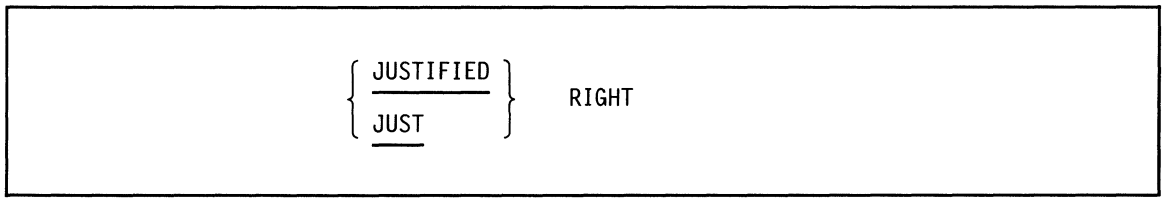
In addition to these restrictions on using HIGH MEMORY, you should be aware of performance degradation when using HIGH MEMORY for programs that are smaller than 1 MD. Less-efficient code is produced when you access the HIGH MEMORY areas due to indexing. Therefore, Unisys recommends that you use HIGH MEMORY only for programs that require addressing data that cannot be accommodated within 1 MD.

Refer to the "File-Control Paragraph" of the "Input-Output Section" in Section 4 for use of the HIGH MEMORY clause with buffers.

JUSTIFIED Clause

The JUSTIFIED clause specifies nonstandard positioning of data within a receiving data item.

The following is the general format:



You can use the JUSTIFIED clause only at the elementary item level. Furthermore, you cannot use it for any data item described as numeric or for which editing is specified.

JUST is an abbreviation for JUSTIFIED.

When a receiving data item is described with the JUSTIFIED clause and the sending data item is larger than the receiving data item, the leftmost characters are truncated. When the receiving data item is described with the JUSTIFIED clause and it is larger than the sending data item, the data is aligned at the rightmost character position in the data item, with space fill for the leftmost character position.

When you omit the JUSTIFIED clause, the standard rules for aligning data within an elementary item apply (see "Data Alignment," earlier in this section).

Level Number

The level number indicates the hierarchy of data within a logical record. In addition, this number identifies entries for Working-Storage items, linkage items, condition names, and the RENAMES clause.

The following is the general format:

```
level number
```

A level number is required as the first element in each data description entry. The level number 01 identifies the first entry in each record description. Level numbers for data description entries subordinate to an FD or SD entry must have the value 01 through 49, 66, or 88. Level numbers for data description entries in the Working-Storage Section must have the values 01 through 49, 66, 77, or 88.

Special level numbers are assigned to certain entries that have no real concept of level:

- Level number 77 identifies noncontiguous Working-Storage data items.
- Level number 66 identifies RENAMES entries.
- Level number 88 defines condition names associated with a conditional variable.

Multiple level 01 entries subordinate to a level indicator (FD, SD, or CD) implicitly redefine the same area.

You can write level numbers in the range 1 through 9 either as a single digit or as a zero followed by a significant digit. At least one space must separate a level number from the following word.

In data description entries that begin with a level number 01 or 77, the level number begins in area A, followed first by its associated record name or item name and and then by appropriate descriptive information.

Successive data description entries can have the same format as the first, or they can be indented according to level number. The entries in the output listing are indented only if the input is indented. Indentation does not affect the magnitude of a level number. When level numbers are to be indented, each new level number can begin any number of spaces to the right of area A. The extent of indentation to the right is determined only by the width of the physical medium; see "Sample Coding Form Usage" in Section 1 for an example of indenting and using level numbers for readability.

OCCURS Clause

The OCCURS clause eliminates the need for separate entries for repeated data items and supplies information required for subscripting or indexing.

The following are the general formats:

Format 1

```
OCCURS integer-2 TIMES

  [ { ASCENDING  } KEY IS data-name-2  [ , data-name-3 ] ... ] ...
    { DESCENDING }

  [ INDEXED BY index-name-1  [ , index-name-2 ] ... ]
```

Format 2

```
OCCURS integer-1 TO integer-2 TIMES

      DEPENDING ON data-name-1

  [ { ASCENDING  } KEY IS data-name-2  [ , data-name-3 ] ... ] ...
    { DESCENDING }

  [ INDEXED BY index-name-1  [ , index-name-2 ] ... ]
```

The OCCURS clause is used in defining tables and other homogeneous sets of repeated data items. Refer to the subsection "Tables" earlier in this section for a complete discussion of tables, subscripting, and indexing.

Whenever the OCCURS clause is used, the data name that is the subject of this entry must be either subscripted or indexed whenever it is referred to in a statement other than SEARCH or USE FOR DEBUGGING. Furthermore, if the subject of this entry is the name of a group item, then all data names belonging to the group must be subscripted or indexed whenever they are used as operands, except as the object of a REDEFINES clause.

Except for the OCCURS clause itself, all data description clauses associated with an item whose description includes an OCCURS clause apply to each occurrence of the item described.

The OCCURS clause cannot be specified in a data description entry that

- Has a level number 01, 66, 77, or 88.
- Describes an item whose size is variable. Its size is variable if the data description of any subordinate item contains Format 2 of the OCCURS clause.

The number of occurrences of the subject entry is defined as follows:

- In Format 1, the value of integer-2 represents the exact number of occurrences.
- In Format 2, the current value of the data item referenced by data-name-1 represents the number of occurrences.

  This format specifies that the subject of this entry has a variable number of occurrences. The value of integer-2 represents the maximum number and the value of integer-1 represents the minimum number of occurrences. This does not imply that the length of the subject of the entry is variable, but that the number of occurrences is variable.

When referencing a group item with a subordinate entry that specifies Format 2 of the OCCURS clause, only that part of the table area that is indicated by the value of data-name-1 is used in the operation.

In Format 2, when a table element is referenced, the value of data-name-1 must fall within the range integer-1 through integer-2, inclusive. If the value of data-name-1 is outside this range, the program is abnormally terminated unless the dollar option NOBOUNDS is set. The value of data-name-1 is used to determine the last table element that can be referenced. When the value of data-name-1 is less than integer-2, the data items whose occurrence numbers exceed the value of data-name-1 are inaccessible. Reducing the value of the data item referenced by data-name-1 has no effect upon the contents of data items whose occurrence numbers exceed the value of the data item referenced by data-name-1. The value of integer-1 must be less than the value of integer-2, and the data description of data-name-1 must describe a positive integer.

A data description entry that contains Format 2 of the OCCURS clause can be followed within that record description only by subordinate data description entries.

In Format 2, the data item defined by data-name-1 must not occupy a character position between the first and last character positions, inclusive, defined by the record description entry containing that OCCURS clause.

If data-name-2 is not the subject of this entry, then

- All of the items identified by the data names in the KEY IS phrase must be within the group item that is subject of this entry.
- Items identified by the data name in the KEY IS phrase must not contain an OCCURS clause.
- No entry can contain an OCCURS clause between the items identified by the data names in the KEY IS phrase and the subject of this entry.

The KEY IS phrase is used to indicate that the repeated data is arranged in ASCENDING or DESCENDING order according to the values contained in data-name-2 and data-name-3. The data-names are listed in descending order of significance.

Data-name-1, data-name-2, and data-name-3 can be qualified.

Data-name-2 must be either the name of the entry containing the OCCURS clause or the name of an entry subordinate to the entry containing the OCCURS clause. Data-name-3 must be the name of an entry subordinate to the group item that is the subject of this entry.

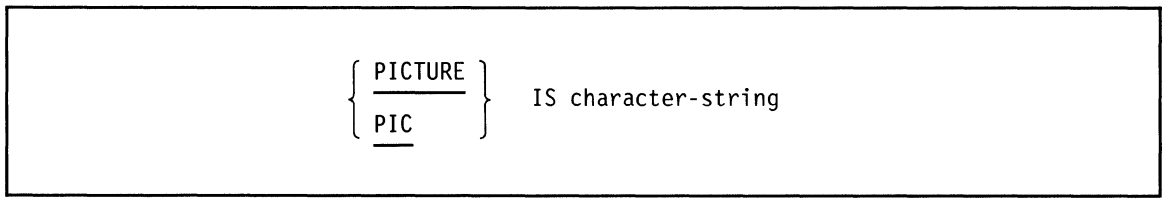
An INDEXED BY phrase is required if the subject of this entry, or an entry subordinate to this entry, is to be referred to by indexing. The index name identified by this clause is not defined elsewhere, since its allocation and format are dependent on the hardware and, not being a data item, it cannot be associated with any data hierarchy. For each index name specified, the compiler assigns a 16-digit area to the associated data block. This area has the construction of a PICTURE S9(7) USAGE COMPUTATIONAL field for the index name value, followed by an 8-digit offset into the table.

Index-name-1 and index-name-2 must be unique words within the program.

PICTURE Clause

The PICTURE clause describes the general characteristics and editing requirements of an elementary item.

The following is the general format:



A PICTURE clause can be specified only at the elementary item level.

PIC is an abbreviation for PICTURE.

A character string consists of certain allowable combinations of characters in the COBOL character set used as symbols. The allowable combinations determine the category of the elementary item.

The maximum number of characters allowed in the character string is 30. This limit is only on the length of the character string itself, not on the length of the data item being described by the character string.

Data Categories

Five categories of data can be described with a PICTURE clause: alphabetic, numeric, alphanumeric, alphanumeric edited, and numeric-edited.

- Use the following rules to define an item as numeric:
  - Its PICTURE character string can only contain the symbols 9, P, S, and V. The number of digit positions that can be described by the PICTURE character string must range from 1 to 18 inclusive.
  - If unsigned, its contents when represented in standard data format must be a combination of the Arabic numerals 0 through 9; if signed, the item can also contain a plus sign (+), minus sign (−), or other representation of an operational sign (see "SIGN Clause" in this section).
- Use the following rules to define an item as alphabetic:
  - Its PICTURE character string can only contain the symbols A and B.
  - Its contents, when represented in standard data format, must be any combination of the 26 letters of the Roman alphabet and the space from the COBOL character set.
- Use the following rules to define an item as alphanumeric:
  - Its PICTURE character string must be certain combinations of the symbols A, X, 9, and the item is treated as if the character string contained all *X*s.

A PICTURE character string that contains all *A*s or all *9*s is not an alphanumeric item.

- Its contents, when represented in standard data format, must be allowable characters in the computer's character set.

- Use the following rules to define an item as alphanumeric edited:
  - Its PICTURE character string must be certain combinations of the symbols A, X, 9, B, 0 (zero), and slash (/).

    The character string must contain at least one B and one X, or at least one 0 (zero) and one X, or at least one / and one X.

    The character string must contain at least one 0 (zero) and one A, or at least one / and one A.
  - The contents, when represented in standard data format, must be allowable characters in the computer's character set.
- Use the following rules to define an item as numeric-edited:
  - Its PICTURE character string must be certain combinations of the symbols B, I, V, Z, 0 (zero), 9, asterisk (*), comma (,), period (.), plus sign (+), minus sign (−), CR, DB, and $. The allowable combinations are determined from the order of precedence of symbols and the editing rules.

    The number of digit positions represented in the PICTURE character string must range from 1 to 18 inclusive.

    The character string must contain at least one B, I, Z, 0 (zero), /, *, +, −, comma (,), period (.), CR, DB, or $.
  - The contents of the character positions of these symbols that are allowed to represent a digit in standard data format must be one of the numerals.

Data Size

Size means the number of character positions occupied by the elementary item in standard data format. It is determined by the number of allowable symbols that represent character positions. An integer that is enclosed in parentheses following the symbols A, B, X, P, Z, 9, 0 (zero), comma (,), *, /, +, −, or $ indicates the number of consecutive occurrences of the symbol. Note that the following symbols can appear only once in a given PICTURE: S, V, period (.), CR, and DB. An integer enclosed in parentheses following the symbol *I* and its associated character represents the number of consecutive occurrences of this I-character pair. For example, *I:(3)* represents *I:I:I:*.

Character String Symbols

The following are the functions of the character string symbols used to describe an elementary item:

A

Each *A* represents a character position that can contain only a letter or a space.

B

Each *B* represents a character position into which the space character is to be inserted.

I

Each *I* indicates that the character immediately following it is treated as a simple insertion character. Specifying the character *I* as the currency symbol overrides its use as a simple insertion indicator. Count the symbol *I* and the character immediately following it as one character in determining the size of the item represented by the PICTURE character string. Include the *I* in the 30-character size limit for the character string appearing in the PICTURE clause.

P

Each *P* indicates an assumed decimal scaling position. It is used to specify the location of an assumed decimal point when the point is not within the number that appears in the data item. Do not count the character *P* in the size of the data item. Scaling position characters are counted in determining the maximum number of digit positions (18) in numeric-edited items or numeric items.

The character *P* can appear only to the left or right as part of a continuous string of Ps within a PICTURE description. The scaling position character *P* implies an assumed decimal point: to the left if Ps are leftmost PICTURE characters and to the right if Ps are rightmost PICTURE characters. Therefore, the assumed decimal point symbol *V* is redundant as either the leftmost or rightmost character within such a PICTURE description.

The character *P* and the period (.) as an insertion character cannot both occur in the same PICTURE character string. Whenever a data item is converted from one form of internal representation to another and that data item is described with the PICTURE character *P*, each digit position described by a *P* is considered to contain the value zero; also the size of the data item is considered to include the digit positions so described.

For example, the picture 9PP represents the numbers 100 through 900, with only the first digit 1 through 9 being stored. The picture PP9 represents the numbers .001 through .009, but with only the last digit 1 through 9 being stored.

S

The letter *S* indicates the presence of an operational sign without necessarily indicating either its representation or position. Write signs as the leftmost character in the PICTURE. Do not count the *S* in determining the size (in terms of standard data format characters) of the elementary item unless the entry is subject to a SIGN clause that specifies the optional SEPARATE CHARACTER phrase. (See "SIGN Clause" in this section.)

When you specify an operational sign for a DISPLAY data item but not a SIGN clause, the sign is maintained and expected in the zone of the most significant (leftmost) character. When the data item is the receiving field in an arithmetic statement, the four zone bits are set to binary 1101 for negative values and to binary 1100 for positive values. When the data item is used in an algebraic comparison or operation to supply an algebraic value, only when the most significant zone is a binary 1101 is the value of the data item considered negative. Only the zone values 1100 and 1101 qualify the data item as NUMERIC if tested by the NUMERIC class condition. For DISPLAY data items, the presence or absence of an operational sign has no effect upon the amount of storage required to contain the data item.

When you specify an operational sign for a COMPUTATIONAL data item but not a SIGN clause, the sign is maintained and expected as a leading separate 4-bit character to the left of the most significant digit position. The binary pattern of the sign character is 1101 for negative values and 1100 for positive values. As in DISPLAY data items, only these values allow the item to be considered NUMERIC in the class condition test. Conversely, specifying an operational sign for COMPUTATIONAL data items increases by one the number of 4-bit character positions occupied by the data item in storage.

V

The *V* indicates the location of the assumed decimal point and can appear only once in a character string. The *V* does not represent a character position. Therefore, it is not counted in the size of the elementary item. When the assumed decimal point is to the right of the rightmost symbol in the string, the *V* is redundant.

X

Each *X* represents a character position that contains any allowable character from the computer's character set.

Z

Each *Z* represents only the leftmost leading numeric character positions that are replaced by a space character when the character in that position is a zero. Count each *Z* in the size of the item.

9

Each *9* represents a character position that contains a numeral. Count the *9* in the size of the item.

0

Each *0* (zero) represents a character position into which the numeral zero is to be inserted. Count the *0* in the size of the item.

Each slash (/) represents a character position into which the stroke character is to be inserted. Count the / in the size of the item.

,

Each comma (,) represents a character position into which the comma character is inserted. Count this character position in the size of the item. The comma (,) insertion character must not be the last character in the PICTURE character string.

The period (.) appears as an editing symbol that represents the decimal point for alignment purposes. It also represents a character position into which the period is to be inserted. Count this character in the size of the item. For a given program, the functions of the period and comma are exchanged if the clause DECIMAL-POINT IS COMMA is stated in the SPECIAL-NAMES paragraph. In this exchange, the rules for the period apply to the comma and the rules for the comma apply to the period wherever they appear in a PICTURE clause. The period used as an insertion character must not be the last character in the PICTURE character string.

+, −, CR, DB

These symbols appear as editing sign control symbols. When used, they represent the character position into which the editing sign control symbol is to be placed. The symbols are mutually exclusive in any one character string. Count each character used in the symbol in determining the size of the item.

*

Each asterisk (*) represents a leading numeric character position into which the asterisk character is placed when the contents of that position are zero. Count each asterisk in the size of the item.

$

The currency symbol ($) represents a character position into which a currency symbol is to be placed. The currency symbol is represented by either $ or by the single currency symbol character specified in the CURRENCY SIGN clause in the SPECIAL-NAMES paragraph. The currency symbol is counted in the size of the item.

Editing Rules

There are two general methods of performing editing in the PICTURE clause: insertion, and suppression with replacement.

Four types of insertion editing are available:

- Simple insertion
- Special insertion
- Fixed insertion
- Floating insertion

There are two types of suppression with replacement editing:

- Zero suppression and replacement with spaces
- Zero suppression and replacement with asterisks

The type of editing that can be performed upon an item is dependent upon the category of the item (see "Data Classification" at the beginning of this section). Table 5–3 shows the type of editing that may be performed upon a given category:

Table 5–3. Data Categories and Editing Methods Allowed

Category	Type of Editing
Alphabetic	Simple insertion B only
Numeric	None
Alphanumeric	None
Alphanumeric-Edited	Simple insertion 0, B, and /
Numeric-Edited	All, †

† Floating insertion and editing by zero suppression with replacement are mutually exclusive in a PICTURE clause. You can use only one type of replacement with zero suppression in a PICTURE clause.

- Simple insertion editing. Use the comma (,), space (B), zero (0), and slash (/) as the insertion characters. In addition, any character that immediately follows the symbol *I* is treated as a simple insertion character. The insertion characters are counted in the size of the item and represent the position in the item into which the character is to be inserted.
- Special insertion editing. The period (.) is used as the insertion character. It also represents the decimal point for alignment purposes. Count the insertion character used for the actual decimal point in the size of the item. The use of the assumed decimal point, represented by the symbol *V*, and the actual decimal point, represented by the period, in the same PICTURE character string is disallowed. The insertion character appears in the item in the same position as shown in the character string.
- Fixed insertion editing. The currency symbol and the editing sign control symbols (+, −, CR, and DB) are the insertion characters. Only one currency symbol and only one of the editing sign control symbols can be used in a given PICTURE character string. When the symbols CR or DB are used, they

represent two character positions in determining the size of the item and they must represent the rightmost character positions that are counted in the size of the item. The symbol + or −, when used, must be either the leftmost or rightmost character position to be counted in the size of the item. The currency symbol must be the leftmost character position, or the second character position when preceded by either + or −, to be counted in the size of the item. Fixed insertion editing results in the insertion character occupying the same character position in the edited item as it occupied in the PICTURE character string. The results produced by the editing sign control symbols depend upon the value of the data item, as depicted in Table 5–4.

Table 5–4. Data Item Values and Results Produced by Editing Sign Control Symbols

Editing Symbol in PICTURE Character String	Result	
	Data Item Positive or Zero	Data Item Negative
+	+	−
−	space	−
CR	2 spaces	CR
DB	2 spaces	DB

- Floating insertion editing. The currency symbol and editing sign control symbols + or − are the floating insertion characters. They are mutually exclusive in a given PICTURE character string.

  Indicate floating insertion editing in a PICTURE character string by using a string of at least two of the floating insertion characters. This string of floating insertion characters can contain any of the fixed insertion symbols or have fixed insertion characters immediately to the right of this string. These simple insertion characters are part of the floating string.

  The leftmost character of the floating insertion string represents the leftmost limit of the floating symbol in the data item. The rightmost character of the floating string represents the rightmost limit of the floating symbols in the data item.

  The second floating character from the left represents the leftmost limit of the numeric data that can be stored in the data item. Nonzero numeric data can replace all the characters at or to the right of this limit.

  In a PICTURE character string, there are only two ways to represent floating insertion editing: represent any or all of the leading numeric character positions on the left of the decimal point by the insertion character; represent all of the numeric character positions in the PICTURE character string by the insertion character.

  If the insertion characters are only to the left of the decimal point in the PICTURE character string, the result is a single floating insertion character placed into the character position immediately preceding either the decimal

point or the first nonzero digit in the data represented by the insertion symbol string, whichever is farther to the left in the PICTURE character string. The character positions preceding the insertion character are replaced with spaces.

If all numeric character positions in the PICTURE character string are represented by the insertion character, the result depends upon the value of the data. If the value is zero, the entire data item contains spaces. If the value is not zero, the result is the same as it is when the insertion character is only to the left of the decimal point.

To avoid being truncated, the PICTURE character string for the receiving data item must be at least the number of characters in the sending data item plus the number of nonfloating insertion characters being edited into the receiving data item, plus one for the floating insertion character.

Zero suppression editing. The suppression of leading zeros in numeric character positions is indicated by the use of the alphabetic character *Z* or the asterisk (*) as suppression symbols in a PICTURE character string. These symbols are mutually exclusive in a given PICTURE character string. Each suppression symbol is counted in determining the size of the item. If *Z* is used, the replacement character is the space; if the asterisk is used, the replacement character is *.

The zero suppression with replacement method is indicated in a PICTURE character string when you use a string of one or more of the allowable symbols to represent leading numeric character positions that are to be replaced when the associated character position in the data contains a zero. Any of the simple insertion characters embedded in the string of symbols, or to the immediate right of this string, are part of the string.

In a PICTURE character string, there are only two ways of representing zero suppression: (1) represent any or all of the leading numeric character positions to the left of the decimal point by suppression symbols; (2) represent all of the numeric character positions in the PICTURE character string by suppression symbols.

If the suppression symbols appear only to the left of the decimal point, any leading zero in the data that corresponds to a symbol in the string is replaced by the replacement character. Suppression terminates at the first nonzero digit in the data represented by the suppression symbol string or at the decimal point, whichever is encountered first.

If all numeric character positions in the PICTURE character string are represented by suppression symbols and the value of the data is not zero, the result is the same as it would be if the suppression characters were only to the left of the decimal point. If the value is zero and the suppression symbol is *Z*, the entire data item is spaces. If the value is zero and the suppression symbol is an asterisk, the data item is filled with asterisks except for the decimal point.

The asterisk, when used as the zero supression symbol, and the clause BLANK WHEN ZERO cannot be used in the same entry.

Table 5-5 shows the results of using various editing characters in picture clauses.

Table 5-5. Editing Application of the PICTURE Clause

Source Area		Receiving Area	
Picture	**Data**	**Editing Picture**	**Edited Data**
9(5)	12345	$ZZ,ZZ9.99	$12,345.00
V9(5)	12345	$$$,$$9.99	$0.12
V9(5)	12345	$ZZ,ZZ9.99	$ 0.12
9(5)	00000	$$$,$$9.99	$0.00
9(3)V99	12345	$ZZ,ZZ9.99	$ 123.45
9(5)	00000	$$$,$$$.$$	
9(5)	01234	$$$,$$$.$$	$1,234.00
9(5)	00000	$**,***.**	*******.**
9(5)	00123	$**,***.**	$***123.**
9(3)V99	00012	$ZZ,ZZ9.99	$ 0.12
9(3)V99	12345	$$$,$$9.99	$123.45
9(3)V99	00001	$ZZ,ZZZ.99	$ .01
9(5)	12345	$$$,$$9.99	$12,345.00
9(5)	00000	$ZZ,ZZZ.ZZ	
9(3)V99	00001	$$$,$$$.$$	$.01
S9(5)	(+)12345	ZZZZ9.99	12345.00+
S9(5)	(−)00123	−99999.99	−00123.00
9(3)V99	12345	999.00	123.00
S9(5)	(−)12345	ZZZZ9.99−	12345.00−
S9(5)	(+)12345	ZZZZ9.99−	12345.00
9(5)	12345	BBB99.99	45.00
S9(5)V	(−)12345	−ZZZZ9.99	−12345.00
S9(5)	(−)12345	$$$$$$.99CR	$12345.00CR
S99V9(3)	(−)12345	−−−.99	−12.34
S9(5)	(+)12345	$$$$$$.99CR	$12345.00
9(3)V99	12345	999.BB	123.
9(5)	12345	00999.00	00345.00

PICTURE Precedence Rules

Figure 5-3 shows the order of precedence when using characters as symbols in a character string. An *X* at an intersection indicates that the symbol or symbols at the top of the column might precede, in a given character string, the symbol or symbols at the left of the row. Arguments appearing in braces indicate that the symbols are mutually exclusive. The currency symbol is indicated by the symbol *cs*.

At least one of the symbols *A*, *X*, *Z*, *9* or * or two of the symbols +, −, or *cs* must be present in a PICTURE string.

Non-floating insertion symbols + and −, floating insertion symbols *Z*, *, +, −, *cs*, and another symbol *P* appear twice in the PICTURE character precedence chart (Table 5–6). The leftmost column and uppermost row for each symbol represents its use on the left of the decimal point position. The second appearance of the symbol in the chart represents its use to the right of the decimal point position.

In the precedence table, an *I* can follow any symbol that a *B* symbol can follow. Furthermore, the *I* character pair can be followed by any symbol that can follow the *B* symbol.

First Symbol / Second Symbol		Non - Floating Insertion Symbols									Floating Insertion Symbols						Other Symbols					
		B or 1	0	/	,	.	{+ -}	{+ -}	{CR DB}	cs	{Z *}	{Z *}	{+ -}	{+ -}	cs	cs	9	A X	S	V	P	P
Non Floating Insertion Symbol	B or 1	x	x	x	x	x	x			x	x	x	x	x	x	x	x	x		x		x
	0	x	x	x	x	x	x			x	x	x	x	x	x	x	x	x		x		x
	/	x	x	x	x	x	x			x	x	x	x	x	x	x	x	x		x		x
	,	x	x	x	x	x	x			x	x	x	x	x	x	x	x			x		x
	.	x	x	x	x		x			x	x		x		x		x					
	{+ -}																					
	{+ -}	x	x	x	x	x				x	x	x			x	x	x			x	x	x
	{CR DB}	x	x	x	x	x				x	x	x			x	x	x			x	x	x
	cs						x															
Floating Insertion Symbol	{Z *}	x	x	x	x		x			x	x											
	{Z *}	x	x	x	x	x	x			x	x	x								x		x
	{+ -}	x	x	x	x					x			x									
	{+ -}	x	x	x	x	x				x			x	x						x		x
	cs	x	x	x	x		x								x							
	cs	x	x	x	x	x	x								x	x				x		x
Other Symbols	9	x	x	x	x	x	x			x	x		x		x		x	x	x	x		x
	A X	x	x	x													x	x				
	S																					
	V	x	x	x	x		x			x	x		x		x		x		x		x	
	P	x	x	x	x		x			x	x		x		x		x		x		x	
	P						x			x									x	x		x

Figure 5-3. PICTURE Character Precedence Chart

REDEFINES Clause

The REDEFINES clause allows the same computer storage area to be described by different data description entries.

The following is the general format:

```
level-number data-name-1  REDEFINES data-name-2
```

Note: *Level-number and data-name-1 are shown in the format above to improve clarity. Level number and data-name-1 are not part of the REDEFINES clause.*

The clause must not be used in level 01 entries in the File Section. In this case, multiple level 01 entries represent implicit redefinitions of the same area.

The REDEFINES clause, when specified, must immediately follow data-name-1. The level numbers of data-name-1 and data-name-2 must be identical, but must not be 66 or 88. Redefinition starts at data-name-2 and ends when a level number less than or equal to that of data-name-2 is encountered.

The REDEFINES clause specifies the redefinition of a storage area, not of the data items occupying the area. Therefore, the usage of data-name-1 need not be the same as the usage of data-name-2, except that DISPLAY or group data items cannot redefine elementary COMPUTATIONAL or INDEX data items that do not begin on a byte boundary. When redefinition occurs at other than the record level (level number 01), the amount of storage allocated for data-name-2 must be the same as the amount of storage implied by the declared size and usage of data-name-1 with the following exceptions:

- A DISPLAY or group data item can redefine an elementary COMPUTATIONAL data item that begins at, but does not end on, a byte boundary if the difference in size can be accounted for by the generation of a 4-bit filler so that the redefining item ends on a byte boundary.
- A DISPLAY or group data item can be redefined by an elementary COMPUTATIONAL data item even though its actual size (including sign position, if described) is one 4-bit character less than the number of 4-bit characters in the storage area. The redefining item is aligned to begin on a byte boundary and end at the middle of the last byte of storage.

Multiple redefinitions of the same character positions are permitted. The entries giving the new descriptions of the character positions must follow the entries defining the area being redefined, without intervening entries that define new character positions. Multiple redefinitions of the same character positions must all use the data name of the entry that originally defined the area.

Multiple level 01 entries subordinate to any given level indicator (FD, SD, or CD) represent implicit redefinitions of the same area.

No entry having a level number numerically lower than the level number of data-name-1 and data-name-2 should occur between the data description entries of data-name-2 and data-name-1.

The entries giving the new description of the character positions must not contain any VALUE clauses, except in condition name entries.

The data description entry for data-name-2 cannot contain a REDEFINES clause. Data-name-2 can be subordinate to an entry that contains a REDEFINES clause.

The data description entry for data-name-2 cannot contain an OCCURS clause. However, data-name-2 can be subordinate to an item whose data description entry contains an OCCURS clause. That is, a table element can be redefined if its description does not include the OCCURS clause. In this case, the reference to data-name-2 in the REDEFINES clause cannot be subscripted or indexed. Neither the original definition nor the redefinition can include an item whose size is defined as variable in the OCCURS clause.

Examples 5–9, 5–10, and 5–11 clarify this last point:

Legal:

```
03  ARRAY-ITEMS OCCURS 10 TIMES.
    05 ARRAY-ITEM PIC X.
    05 ARRAY-ITEM-NR REDEFINES ARRAY-ITEM PIC 9.
```

Example 5–9. Legal Array Definition

Legal:

```
03  ARRAY-ITEMS-HOLD.
    05 ARRAY-ITEMS OCCURS 10 TIMES
       07 ARRAY-ITEM PIC X.
03  ARRAY-RDF REDEFINES ARRAY-ITEMS-HOLD.
    05 ARRAY-ITEM-NR OCCURS 10 TIMES PIC 9.
```

Example 5–10. Legal Array Definition

Illegal:

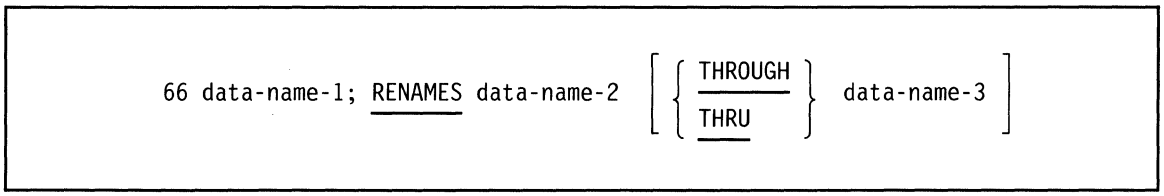
```
03  ARRAY-ITEMS OCCURS 10 TIMES PIC X
03  ARRAY-ITEM-HOLD REDEFINES ARRAY-ITEMS PIC X (10).
```

Example 5–11. Illegal Array Definition

RENAMES Clause

The RENAMES clause permits alternative, possibly overlapping, groupings of elementary items.

The following is the general format:



Note: *Level number 66 and data-name-1 are shown in the above format to improve clarity. They are not part of the RENAMES clause.*

Data-name-1 assumes all the characteristics of data-name-2, as determined from the data description of data-name-2, including usage, justification, synchronization, and editing requirements.

You can write one or more RENAMES entries for a logical record. All RENAMES entries referring to data items within a given logical record must immediately follow the last data description entry of the associated record description entry (see Example 5–12).

Data-name-2 and data-name-3 must be names of elementary items or groups of elementary items in the same logical record; they cannot be the same data name. A level 66 entry cannot rename another level 66 entry nor can it rename a level 77, 88, or 01 entry.

You cannot use data-name-1 as a qualifier. It can be qualified only by the names of the associated level 01, FD, or SD entries. Neither data-name-2 nor data-name-3 can have an OCCURS clause in its data description entry or can be subordinate to an item that has an OCCURS clause in its data description entry.

Data-name-2 and data-name-3 can be qualified.

No item within the range, including data-name-2 and data-name-3, if specified, can be an item whose size is defined as variable in the OCCURS clause.

When you do not specify data-name-3, data-name-2 can be either a group or an elementary item:

- When it is a group item, data-name-1 is treated as a group item
- When it is an elementary item, data-name-1 is treated as an elementary item.

When you do specify data-name-3, the following are the rules:

- If data-name-2 is an elementary item and data-name-3 is an elementary item, data-name-1 is a group item that includes all elementary items starting with data-name-2 and concludes with data-name-3.
- If data-name-2 is an elementary item and data-name-3 is a group item, data-name-1 is a group item that includes all elementary items starting with data-name-2 and concludes with the last elementary item in data-name-3.
- If data-name-2 is a group item and data-name-3 is an elementary item, data-name-1 is the first elementary item in data-name-2 and concludes with data-name-3.
- If data-name-2 is a group item and data-name-3 is a group item, data-name-1 is the first elementary item in data-name-2 and concludes with the last elementary item in data-name-3.
- If data-name-2 is an elementary COMPUTATIONAL or INDEX data item, then data-name-3 must be positioned to begin at an 8-bit character boundary.
- If data-name-3 is an elementary COMPUTATIONAL or INDEX data item, it must be positioned to end at the end of an 8-bit character boundary.

The beginning of the area described by data-name-3 must not be to the left of the beginning of the area described by data-name-2. The end of the area described by data-name-3 must be to the right of the end of the area described by data-name-2. Data-name-3, therefore, cannot be subordinate to data-name-2.

Example 5–12 shows the use of the RENAMES Clause.

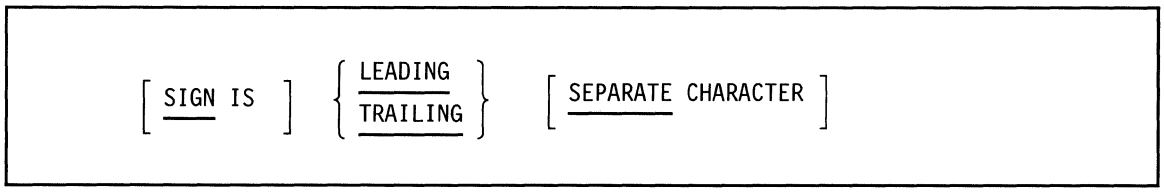
```
01  TAB.
    03  A.
        05  A1      PIC X.
        05  A2      PIC XXX.
        05  A3      PIC XX.
        05  A4      PIC XX.
    03  B.
        05  B1      PIC XX.
        05  B2      PIC X(6).
        05  B3      PIC X(8).
66  C RENAMES A.                *(THIS RENAMES A1 THRU A4)
66  D RENAMES A.                *(THIS RENAMES A1 THRU A4)
66  E RENAMES A1 THRU A3.
66  F RENAMES A4 THRU B2.
66  G RENAMES A2 THRU B.        *(THIS RENAMES A2 THRU B3)
66  H RENAMES A THRU B.         *(THIS RENAMES A1 THRU B3)
```

Example 5–12. Examples of RENAMES

SIGN Clause

Use the SIGN clause to specify the position and the mode of representation for the operational sign when it is necessary to describe these properties explicitly.

The following is the general format:



The optional SIGN clause, if present, specifies the position and the mode of representation of the operational sign for the numeric data description entry to which it applies, or for each numeric data description entry subordinate to the group to which it applies. The SIGN clause applies only to numeric data description entries whose PICTURE contains the character *S*, which indicates the presence of the operational sign.

Describe the numeric data description entries to which the SIGN clause applies as USAGE, DISPLAY, or COMPUTATIONAL.

At most, one SIGN clause can apply to any given numeric data. If the SIGN clause is specified in the data description entry for a group item, then you must not use it in any subordinate data description.

If the CODE-SET clause is specified, you must describe any signed numeric data description entries associated with that file description entry with the SIGN IS SEPARATE clause.

Every numeric data description entry whose PICTURE contains the character *S* is a signed numeric data description entry. If a SIGN clause applies to such an entry and conversion is necessary for computation or comparisons, conversion takes place automatically.

The default position and representation of operational signs applies when a numeric data description entry whose PICTURE contains the character *S* but which has no optional SIGN clause. (see "Character String Symbols," under "PICTURE Clause" earlier in this section).

If a SIGN clause without a SEPARATE CHARACTER phrase applies to a numeric data description entry, then the following are relevant for EBCDIC native character sets:

- If the usage of the data item is DISPLAY, the operational sign is maintained and expected as a binary 1100 or 1101 in the zone of the LEADING or TRAILING character. No additional storage is allocated for the data item.
- If the usage of the data item is COMPUTATIONAL, the operational sign is maintained and expected as a binary 1100 or 1101 LEADING or TRAILING character, increasing by one 4-bit character the amount of storage allocated for the data item beyond the amount that would be allocated for an unsigned COMPUTATIONAL data item. The presence or absence of the SEPARATE CHARACTER phrase has no effect upon the position or representation of the operational sign for COMPUTATIONAL data items.

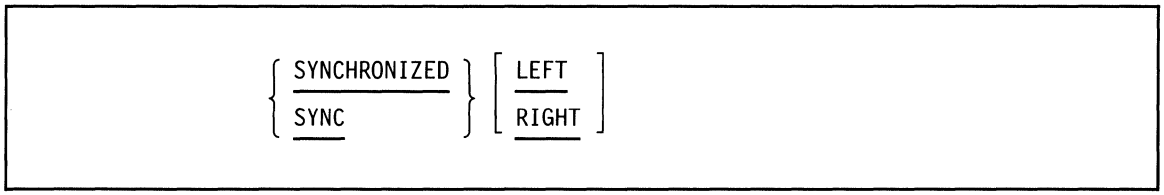
If a SIGN clause with a SEPARATE CHARACTER phrase applies to a numeric data description entry, then the following is relevant:

- If the usage of the data item is DISPLAY, the operational sign is maintained and expected as a LEADING or TRAILING character separate from, and in addition to, the numeric character positions. The operational sign for negative values is the minus sign (−) character and the sign for positive values is the plus sign (+) character .
- If the usage of the data item is COMPUTATIONAL, the rules listed in item 2 of the preceding list apply.

SYNCHRONIZED Clause

The SYNCHRONIZED clause specifies the alignment of an elementary item on the natural boundaries of computer memory.

The following is the general format:



This clause can appear only with an elementary item.

SYNC is an abbreviation for SYNCHRONIZED.

This clause specifies that the subject data item is to be aligned in the computer to a byte boundary. If the previous data item did not end on a byte boundary, the computer generates an implicit 4-bit FILLER. This unused filler digit is included in the following:

- The size of any group item(s) to which the elementary item belongs
- Any redefinition of the previous data item when it is the object of a REDEFINES clause (see "REDEFINES Clause" and "USAGE Clause" in this section)

Whenever a SYNCHRONIZED item is referenced in the source program, the original size of the item, as shown in the PICTURE clause, is used in determining any action that depends on size, such as justification, truncation, or overflow.

Either RIGHT or LEFT following SYNCHRONIZED has no effect on the positioning of the data item. RIGHT and LEFT are handled as comment entries. If the data description of an item contains the SYNCHRONIZED clause and an operational sign, the sign of the item appears in the normal operational sign position, regardless of whether the item is SYNCHRONIZED LEFT or SYNCHRONIZED RIGHT.

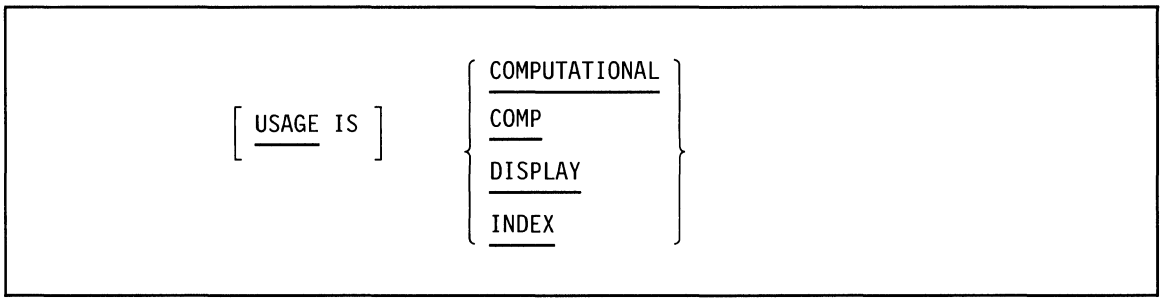
When the SYNCHRONIZED clause is specified in a data description entry of a data item that contains or is subordinate to anything that contains an OCCURS clause, the following is true:

- Each occurrence of the data item is SYNCHRONIZED.
- Any implicit FILLER generated for other data items within that same table is generated for each occurrence of those data items.

USAGE Clause

The USAGE clause specifies the format of a data item in computer storage.

The following is the general format:



This clause does not affect the use of the data item in computer storage, although the specifications for some statements in the Procedure Division can restrict the USAGE clause of the corresponding operands. The USAGE clause can affect the type of character representation of the item.

You can write the USAGE clause in any data description entry with a level number other than 66 or 88. If you write the USAGE clause at a group level, it applies to each elementary item in the group. Therefore, if you write the USAGE in the data description entry for a group item, you can also write it in the data description entry for any subordinate elementary item or group item. However, both entries must specify the same USAGE. Items of different USAGE can appear in the same record.

A COMPUTATIONAL item can represent a value to be used in computations. It can also contain hexadecimal values ranging from @A@ through @F@; if so, it should not be used in arithmetic operations. The PICTURE character string of a COMPUTATIONAL item can contain only the characters 0 (zero) through 9, the operational sign character *S*, the implied decimal point character *V*, and/or one or more *P* characters.

If a group item is described as COMPUTATIONAL, the elementary items in the group are COMPUTATIONAL. The group item itself is not COMPUTATIONAL. This item is a group data item whose class is alphanumeric and whose usage is DISPLAY. It can be referenced any place in the syntax. The size of the group item is considered to be one character for every two 4-bit digits forming the item.

Elementary COMPUTATIONAL data items are represented internally as contiguous 4-bit digits.

COMP is an abbreviation for COMPUTATIONAL.

The usage is implicitly DISPLAY if the USAGE clause is not specified for either an elementary item or for any group to which the item belongs. DISPLAY data items are represented internally as contiguous 8-bit characters from the EBCDIC character set.

The USAGE IS DISPLAY clause indicates that the format of the data is a standard data format.

Every occurrence of a DISPLAY data item begins and ends on a byte boundary. Within a record description, the declaration of a DISPLAY data item immediately following a COMPUTATIONAL or INDEX data item that does not end on a byte boundary causes automatic generation of a 4-bit filler between the two items. This filler area between the two data items is not included in the size of either item; it is included in the size of all group items to which the two items are subordinate. Similarly, if the last item declared within a group item at the next lower hierarchical level is a COMPUTATIONAL or INDEX data item that does not end on a byte boundary, automatic generation of a 4-bit filler occurs. This filler is included in the size of the group item.

An elementary item described with the USAGE IS INDEX clause is called an index data item. It contains a value that must correspond to an occurrence number of a table element. The elementary item cannot be a conditional variable.

If a group item is described with the USAGE IS INDEX clause, the elementary items in the group are index data items. The group itself is not an index data item, so you cannot use it in the SEARCH or SET statement or in a relation condition. This group item is considered a group data item whose class is alphanumeric and whose usage is DISPLAY; it can be referenced at any acceptable place in the syntax. The size of the group item is considered in terms of DISPLAY characters, four characters for each subordinate index data item.

An index data item can be part of a group that is referred to in a MOVE or an input/output statement. Consequently no conversion takes place.

An index data item can contain a signed value. An index data item occupies the same space and has the same alignment as an item declared PICTURE S9(7) USAGE IS COMPUTATIONAL.

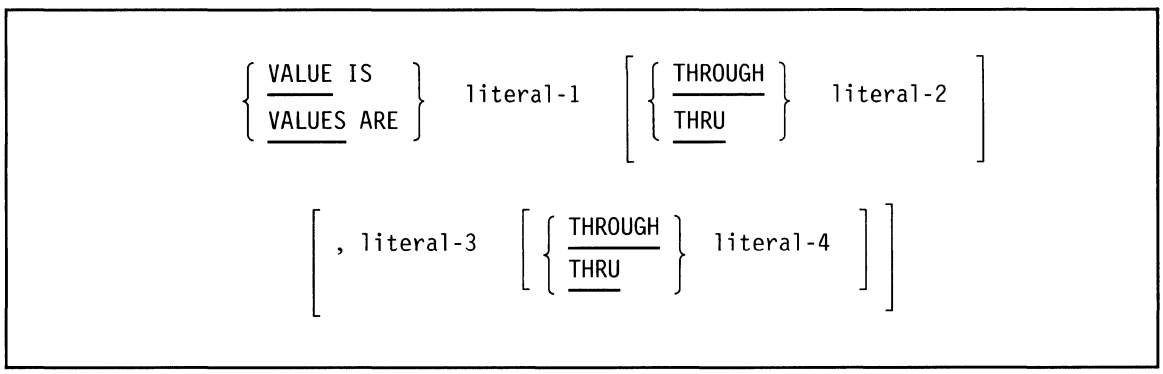
An index data item can be referenced explicitly only in a SEARCH statement, SET statement, or relation condition.

You cannot use the SYNCHRONIZED, JUSTIFIED, PICTURE, VALUE, and BLANK WHEN ZERO clauses to describe group or elementary items described with the USAGE IS INDEX clause.

VALUE Clause

The VALUE clause defines the initial value of Working-Storage items and the values associated with a condition name.

The following is the general format:



In a condition name entry, the VALUE clause is required. The VALUE clause and the condition name itself are the only two clauses permitted in the entry. The characteristics of a condition name are implicitly those of its conditional variable.

Literal-2, literal-3, and literal-4 can be used only in connection with condition names. Wherever the THRU phrase is used, literal-1 must be less than literal-2, and literal-3 must be less than literal-4.

A figurative constant can be substituted where a literal is specified.

Rules governing the use of the VALUE clause differ with the respective sections of the Data Division:

- In the File Section, the VALUE clause can be used only in condition name entries.
- In the Working-Storage Section the VALUE clause must be used in condition name entries. The VALUE clause can also be used to specify the initial value of any other data item, in which case the clause causes the item to assume the specified value at the start of the object program. If the VALUE clause is not used in an item's description, the initial value is undefined.

The VALUE clause must not conflict with other clauses in the data description of the item or in the data description within the hierarchy of the item. The following rules apply to the VALUE clause:

- If the item category is numeric, all literals in the clause must be numeric. If the literal defines the value of a Working-Storage item, the literal is aligned in the data item according to standard alignment rules (see "Data Alignment" earlier in this section).
- If the category of the item is alphabetic, alphanumeric, alphanumeric-edited, or numeric-edited, all literals in the VALUE clause must be nonnumeric literals. The literal is aligned in the data item as if the data item had been described as alphanumeric (see "Data Alignment" in this section). Editing characters in the PICTURE clause are included in determining the size of the data item, but have no effect on initialization of the data item. Therefore, the VALUE for an edited item is presented in an edited form.
- If any BLANK WHEN ZERO or JUSTIFIED clause is in effect, initialization is not affected.

Note: *The figurative constant ZERO belongs to the category numeric. The figurative constant SPACE belongs to the category alphabetic. All other figurative constants (such as HIGH-VALUES and LOW-VALUES) belong to the category alphanumeric. Refer to "Figurative Constants" in Section 2.*

All numeric literals in a VALUE clause of an item must have a value that is within the range indicated by the PICTURE clause and must not have a value that requires truncation of nonzero digits. Nonnumeric literals in a VALUE clause of an item must not exceed the size indicated by the PICTURE clause.

A signed numeric literal must be associated with a signed numeric PICTURE character string.

The VALUE clause must not be stated in a data description entry that contains or is subordinate to an entry that contains an OCCURS clause. This rule does not apply to condition name entries.

The VALUE clause cannot be stated for any items whose size is variable (see "OCCURS Clause" in this section).

The VALUE clause must not be stated in a data description entry that contains or is subordinate to an entry that contains a REDEFINES clause. This rule does not apply to condition name entries.

If the VALUE clause is used in an entry at the group level, the literal must be a figurative constant or a nonnumeric literal. Also, the group area is initialized without consideration for the individual elementary or group items contained within this group. The VALUE clause cannot be stated at the subordinate levels within this group.

The VALUE clause must not be written for a group containing items for which USAGE (other than USAGE IS DISPLAY) is specified either explicitly or implicitly.

Working-Storage Section

The Working-Storage Section is composed of the section header, followed by data description entries for noncontiguous data items and/or record description entries. Each Working-Storage Section record name and noncontiguous item name must be unique since it cannot be qualified. Subordinate data names need not be unique if they can be made unique by qualification.

Noncontiguous Working-Storage

In Working-Storage, items and constants that bear no hierarchical relationship to one another need not be grouped into records, provided they need no further subdividing. Instead, they are classified and defined as noncontiguous elementary items. Each of these items is defined in a separate data description entry that begins with the special level number 77.

The following is the general format:

```
                    { The PICTURE clause          }
77 data-name        {                             }
                    { The USAGE IS INDEX clause   }

          [Any other data description clauses]
```

Working-Storage Records

In Working-Storage, you must group data elements and constants that bear a definite hierarchic relationship to one another into records according to the rules for forming record descriptions. You can use all clauses used in record descriptions in the File Section in record descriptions in the Working-Storage Section (see Example 5–13).

Initial Values

In the Working-Storage Section, you specify the initial value of any item except an index data item by using the VALUE clause with the data item. The initial value of any index data item is unpredictable. The initial value of any non-index data item without a VALUE clause is unpredictable.

Coding Example

Example 5–13 shows how you can code the Working-Strorage section.

```
WORKING-STORAGE SECTION.
77  MASTER-KEY              PIC 9(8)    COMP.
77  TOTAL-SALES             PIC 9(10)   COMP    VALUE ZEROS.
01  STATE-TABLE.
    03  STATES.
        05  CALIF           PIC 9(5).
        05  MICH            PIC 9(5).
        05  FLORIDA         PIC 9(5).
        05  ARIZONA         PIC 9(5).
    03  STATE-KEY  REDEFINES  STATES    OCCURS  4 TIMES.
        05  STATE-CODE      PIC 99.
        05  COUNTY          PIC 99.
        05  CAPITAL-CODE    PIC 9.
01  HEADING-LINE.
    03  FILLER              PIC X(52)   VALUE SPACES.
    03  FILLER              PIC X(17)   VALUE
            "SALES PERFORMANCE".
    03  FILLER              PIC X(10)   VALUE SPACES.
    03  PAGE-NO             PIC 9999.
    03  FILLER              PC  X(49)   VALUE SPACES.
```

Example 5–13. Coding the Working-Storage Section

Linkage Section

The Linkage Section in a program is pertinent only if the object program is to function under the control of a CALL statement that contains a USING phrase.

The Linkage Section describes data available through the calling program, but referred to in both the calling and called program. No space is allocated in the program for data items referenced by data names in the Linkage Section of that program. Procedure Division references to these data items are resolved at program execution by equating the reference in the called program to the location used in the calling program. No correspondence is established for index names; index names in the calling and called programs always refer to separate indices.

Data items defined in the Linkage Section of the called program can be referenced within the Procedure Division of the called program only under certain conditions: if they are either specified as operands in the USING phrase of the Procedure Division statement or are subordinate to the specified operands, and the program is under control of a CALL statement that specifies a USING phrase.

The structure of the Linkage Section is the same as that described in the Working-Storage Section; the structure begins with a section header, followed by Data Description entries for noncontiguous data items and/or Record Description entries.

Each Linkage Section record name and noncontiguous item name must be unique within the called program, since the names cannot be qualified.

Among the items defined in the Linkage Section, only data-name-1, data-name-2, ... data-name-n in the USING phrase of the Procedure Division statement, data items subordinate to these data names, and condition names and/or index names associated with either the data names and/or subordinate data items, can be referenced.

Inter-Program Communication

Items in the Linkage Section that have no hierarchical relationship to each other need not be grouped into records. They are classified and defined as noncontiguous elementary items. Each of these items is defined in a separate Data Description entry that begins with level number 77.

The following data clauses are required in each Data Description entry:

- Level number 77
- Data name
- PICTURE clause or USAGE IS INDEX clause

Other Data Description clauses are optional. They can be used to complete the description of the necessary item.

Section 6
Procedure Division

This section describes the elements that can be included in the fourth and last division of a COBOL program, the Procedure Division.

General

Declaratives must be grouped at the beginning of the Procedure Division, preceded by the keyword DECLARATIVES and followed by the keywords END DECLARATIVES. The rest of the Procedure Division contains the nondeclarative procedures. The end of the Procedure Division and the physical end of the program is that physical position in a COBOL source program after which no further procedures appear.

A procedure is composed of a paragraph, or group of successive paragraphs, or a section, or a group of successive sections, within the Procedure Division. If one paragraph is in a section, then all paragraphs must be in sections. A procedure-name is a word used to refer to a paragraph or section in the source program in which it occurs. The procedure-name consists of a paragraph-name (which can be qualified) or a section-name.

Declaratives are procedures that operate under the control of the input/output system. Declaratives consist of compiler-directing sentences and their associated procedures. When declaratives are used, they must be grouped together at the beginning of the Procedure Division. The group of declaratives must be preceded by the keyword DECLARATIVES, starting in area A of the coding form, and must be followed by the words END DECLARATIVES, also starting in area A. The next source statement following the END DECLARATIVES statement must be a section-name. Each declarative consists of a single section. The USE statement is the only declarative statement. Complete rules can be found under "USE Statement" in this section.

DECLARATIVES cannot be declared for port files.

A SORT statement cannot appear in the DECLARATIVES section of a COBOL program.

A section consists of a header followed by zero or more successive paragraphs. A section header consists of a section-name, followed by the word SECTION, optionally followed by a segment-number.

The section-name is any unique user-defined word. It need not contain any alphabetic characters. All section headers must begin in area A of the coding form (see "Coding Form" in Section 1). A section ends immediately before the next section, at the end of the Procedure Division, or in the declaratives portion of the Procedure Division at the keywords END DECLARATIVES. Dividing the program is optional, but such divisions are useful for segmentation.

A paragraph consists of a paragraph-name followed by a period, a space, and zero or more successive sentences. A paragraph-name is any user-defined word that is unique or can be made unique by qualification (using the section-name and not the word *SECTION*). A program, therefore, can contain more than one paragraph with the same name provided each is in a different section. A paragraph-name need not be qualified when referred to from within the same section and need not contain any alphabetic characters. All paragraph-names must begin in area A of the coding form (see "Coding Form" in Section 1). A paragraph ends immediately before the next paragraph-name or section-name or at the end of the Procedure Division or, in the declaratives portion of the Procedure Division, at the keywords END DECLARATIVES.

A statement is a syntactically valid combination of words and symbols beginning with a COBOL verb.

A sentence consists of one or more statements and ends with a period.

An identifier is the word or words that make unique reference to a data item.

Execution begins with the first statement of the Procedure Division, excluding declaratives. Statements are then executed in the order in which they are presented for compilation, except where the rules indicate some other order.

Structure

The Procedure Division is identified by and must begin with the following header:

```
PROCEDURE DIVISION  [ USING  data-name-1, [data-name-2] ... ].
```

Include the USING phrase only if the program is to function under the control of a CALL statement and the CALL statement in the calling program also contains a USING phrase.

Each operand in the USING phrase of the Procedure Division statement must be defined as a data item in the Linkage Section of the respective program.

Linkage data items are processed according to their data descriptions given in the called program.

When specified, data-name-1 in the called program and data-name-1 in the calling program both refer to the same data. Furthermore, data-name-2 in both the called and calling programs also refer to the same data. Each data description must define an equal number of characters; however, the data-name in the called program and in the calling program need not be the same name. When specified, a file-name must be defined as an 8 digit data-item. Also, each data-name cannot be specified more than once in a USING phrase. However, you can specify each data-name more than once in the USING phrase of a CALL statement.

The body of the Procedure Division must conform to one of the following formats:

Format 1

```
[DECLARATIVES.
{section-name SECTION [segment-number]. declarative-sentence
[paragraph-name. [sentence] ...] ...} ...
END DECLARATIVES.]
{section-name SECTION [segment-number].
[paragraph-name. [sentence] ...] ...} ...
```

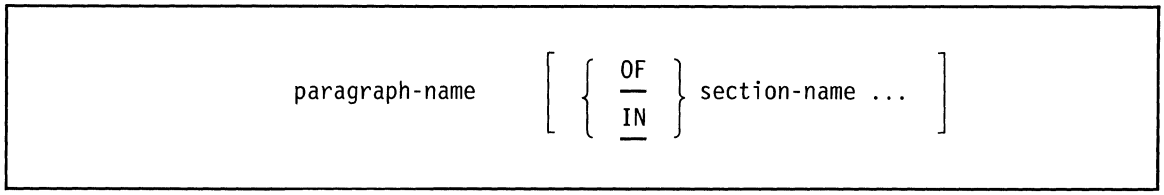
Format 2

```
{paragraph-name. [sentence] ...} ...
```

Qualification

In the Procedure Division, you can use qualification to make unique procedure names. Qualification is performed by following a paragraph name with a qualifier consisting of a section name preceded by either IN or OF. The words IN and OF are logically equivalent.

The following is the general format for paragraph-name qualification:



The rules for paragraph name qualification are as follows:

- Do not duplicate a paragraph name within a section.
- A section name is the only qualifier available for a paragraph name.
- Do not include the word *Section* when qualifying a paragraph name with a section name.
- You do not need to qualify a paragraph when it is referred to from within the same section.
- No name can be both a data name and a procedure name.
- You can qualify a name even though it does not need qualification. If there is more than one combination of qualifiers that ensures uniqueness, then any such set can be used. The complete set of qualifiers for a data name must not be the same as any partial set of qualifiers for another data name.

Segmentation

COBOL segmentation provides a means of specifying object program overlay requirements. COBOL segmentation deals only with the segmentation of procedures, not of data. As such, only the Procedure Division and the Environment Division are considered in determining segmentation requirements for an object program.

Although it is not mandatory, the Procedure Division for a source program is usually written as a group of sections, each of which is composed of a series of closely related operations that are designed collectively to perform a particular function.

When segmentation is used, the entire Procedure Division must be in sections. In addition, each section must be classified as belonging either to the fixed portion or to one of the independent segments of the object program. Segmentation in no way affects the need to qualify procedure-names to ensure uniqueness.

The segmentation facilty permits the user to physically subdivide the Procedure Division of a COBOL object program. All source paragraphs that contain the same segment-number in their section headers are considered at object time to be one segment. Segment-numbers can range from 0 through 9999 ; however, the total number of segments in any object program is limited to 999

Program segments can be of two types: fixed or independent, as determined by the programmer's assignment of segment-numbers.

Fixed Segments

Fixed segments are defined as the part of the object program that is logically treated as if it were always in memory. This portion of the program is composed of two types of segments: fixed permanent segments (segment numbers blank or 0 up to the segment limit) and fixed overlayable segments (the segment limit through 49).

Fixed permanent segments are always in computer storage during the execution of the entire program. They cannot be overlayed except when the system is executing another program. In this case, fixed segments can be "rolled out" temporarily. They cannot be overlayed by any other part of the program. You can vary the number of fixed permanent segments in the fixed portion by using a special facility called the SEGMENT-LIMIT clause (see "SEGMENT-LIMIT Clause" in Section 4).

A fixed overlayable segment is a segment in the fixed portion of the program that, though logically treated as if it were always in memory, can be overlaid by another segment to optimize memory utilization.

Such a segment, if called for by the program, is always made available in its last-used state. Fixed overlayable segments can be overlayed during program execution. However, any such overlaying is transparent to the user. These segments are logically equivalent to fixed segments, but physically different from them. Create fixed overlayable segments using the SEGMENT-LIMIT clause.

Independent Segments

An independent segment is defined as a part of the object program that can overlay, and be overlaid by, another independent segment. An independent segment is specified by segment-numbers ending in 050 through 999.

Independent segments can be organized into overlay areas. (See the following discussion of "Overlay Areas".)

The ALTER statement cannot reference a paragraph in an independent segment unless the ALTER statement resides in an independent segment that has the same segment-number as that of the paragraph it references. For complete information, refer to "ALTER Statement" in this section.

Segmentation Logic

Sections to be segmented are classified, by use of a system of segment-numbers and the following criteria:

- Logic requirements. Sections that must be available for reference at all times or that are referred to very frequently are normally classified as belonging to one of the permanent segments; sections that are used less frequently are normally classified as belonging either to one of the overlayable fixed segments or to one of the independent segments, depending on logic requirements.
- Frequency of use. Generally, the more frequently a section is referred to, the lower its segment-number; the less frequently it is referred to, the higher its segment-number.
- Relationship to other sections. Sections that frequently communicate with one another should be given the same segment-numbers or should reside in different overlay areas.

The logical sequence of the program is the same as the physical sequence except for specific transfers of control. Control can be transferred within a source program to any paragraph in a section; that is, it is not mandatory to transfer control to the beginning of a section. The CALL MODULE statement is an exception. It transfers control to the first statement in a section.

Segment-Numbers

Section classification is accomplished by means of a system of segment-numbers. The segment-number is included in the section header.

The following is the general format:

```
section-name SECTION [segment-number]
```

All sections that have the same segment-number constitute a program segment. Sections with the same segment-numbers need not be physically contiguous in the source program.

The segment-number must be a blank or an integer ranging in value from 0 through 9999 . Segments with no segment-number or with segment-numbers ending in 000 through 049 belong to the fixed portion of the object program. These are all fixed permanent segments unless you use the SEGMENT-LIMIT clause. (The default value of the SEGMENT-LIMIT clause is 049.) Segments with segment-numbers ending in 050 through 999 are independent segments. If the segment-number contains four digits, the first digit designates an overlay area and the last three digits determine whether the segment is fixed or independent.

If the segment-number is omitted from the section header, the segment-number is assumed to be zero.

Sections in the declaratives must contain segment-numbers less than 50.

Overlay Areas

A program with fixed overlayable or independent segments always has at least one overlay area. This area is occupied by the latest overlay, that is, the fixed overlayable segment or the independent segment used most recently. When control is transferred to a different overlay, the new segment replaces the segment previously occupying its overlay area. Therefore, you can save memory space by declaring sections to be fixed overlayable or independent segments; the overlay area is only as large as the largest fixed overlayable or independent segment it must contain.

Since program processing is interrupted while an overlayable segment is brought into memory, using overlayable segments forces a compromise between program size and execution speed. Seldom called segments should be overlayable, while frequently called segments should remain in main memory.

In order to permit more than one overlayable segment to reside in memory at one time and thereby reduce execution time when there are frequent calls to or between these segments, you can organize overlayable segments into multiple overlay areas. The following rules apply to overlay areas:

- The high-order digit of a 4-digit section number specifies the overlay area into which that segment can be placed.
- Each overlay area is the size of the largest segment assigned to it.
- At any time, the overlay area is occupied only by the last segment brought into it.
- When control passes to a segment not in memory, that segment overlays the segment previously occupying its overlay area; the contents of other overlay areas are unchanged.

The dollar option MAP shows the layout of overlays and their sizes on the program output listing. Figure 6–1 illustrates the memory utilization of a program with several overlay areas.

Sections numbered 1022 and 1023 are gathered into segments that can only occupy overlay area 1. Sections 2032, 2033, and 2040 are gathered into segments that can only occupy overlay area 2. Overlay area 3 contains the segments consisting of sections numbered 3071 and 3082. Since Figure 6–1 shows overlayable segments that are numbered 1022, 1023, 2032, 2033, and 2040, you can assume that the SEGMENT-LIMIT clause of the program was used to declare a lower limit of 21. If this clause had not been specified, these segments would have been fixed-permanent segments and would have been resident (nonoverlayable) because the default SEGMENT-LIMIT is 049.

Each overlay area is occupied by only one of its segments at a time. (Note that overlay area 2 is larger than the sum of the sizes of the segments formed from sections 2032 and 2040. Nevertheless, only one of those segments can occupy that area at any one time.) If the destination segment is present in its area from a previous use, this arrangement allows transfer of control from a segment in one overlay area to a segment in another overlay area without interrupting the program for a physical overlay.

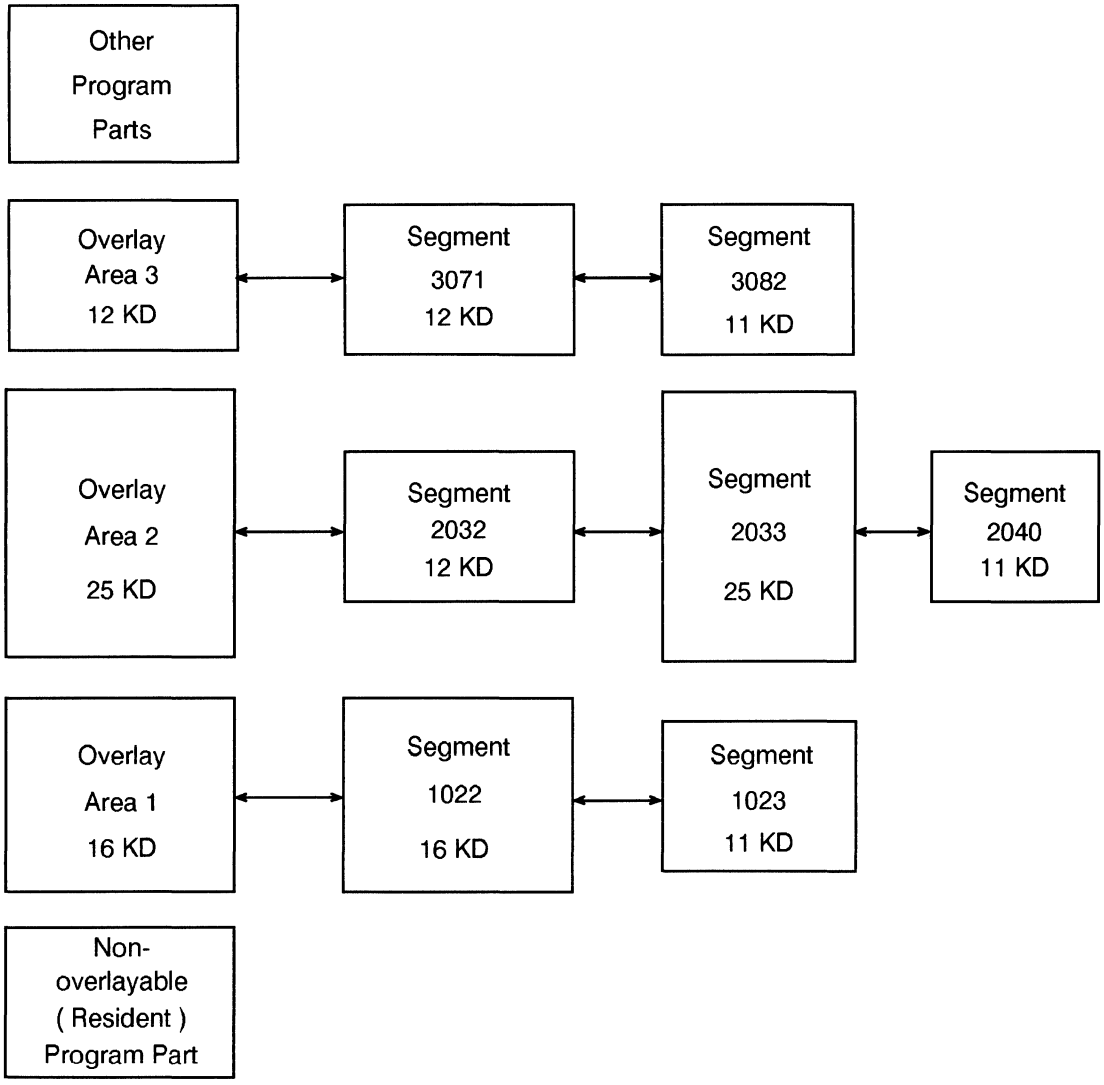


Figure 6–1. Example of a Program with Multiple Overlay Areas

Statements and Sentences

There are three types of statements and sentences: conditional, compiler-directing, and imperative.

Conditional Statements and Sentences

A conditional statement specifies that the truth value of a condition is to be determined and that the subsequent action of the object program is dependent on this truth value.

A conditional statement is any one of the following:

- An IF, SEARCH, or RETURN statement
- A READ statement that specifies the AT END or INVALID KEY phrase
- A WRITE statement that specifies the INVALID KEY or END-OF-PAGE phrase
- A START, REWRITE, or DELETE statement that specifies the INVALID KEY phrase
- An arithmetic statement (ADD, COMPUTE, DIVIDE, MULTIPLY, SUBTRACT) that specifies the SIZE ERROR phrase
- A STRING or UNSTRING statement that specifies the ON OVERFLOW phrase

Compiler-Directing Statements and Sentences

A compiler-directing statement causes the compiler to take a specific action during compilation. A compiler-directing statement consists of a compiler-directing verb and its operands. The compiler-directing verbs are COPY and USE.

A compiler-directing sentence is a single compiler-directing statement terminated by a period.

Imperative Statements and Sentences

An imperative statement indicates a specific unconditional action that the object program is to take. An imperative statement is any statement that is neither a conditional statement nor a compiler-directing statement. An imperative statement can consist of a sequence of imperative statements, each of which is separated from the next by a separator. The following are the imperative verbs:

ACCEPT	INSPECT	SEEK
ADD[1]	MERGE	SEND[6]
ALTER	MOVE	SET
CALL	MULTIPLY[2]	SORT
CLOSE	OPEN	START[2]
COMPUTE[1]	PERFORM	STOP
DELETE[2]	READ[4]	STRING[3]
DISPLAY	RECEIVE[6]	SUBTRACT[1]
DIVIDE[1]	RELEASE	UNSTRING[3]
EXIT	REWRITE[2]	WRITE[5]
GO		

1 Without the optional SIZE ERROR phrase
2 Without the optional INVALID KEY phrase
3 Without the optional ON OVERFLOW phrase
4 Without the optional AT END phrase or INVALID KEY phrase
5 Without the optional INVALID KEY phrase or END-OF-PAGE phrase
6 Without the optional ON EXCEPTION phrase

When *imperative-statement* appears in the general format of statements, it refers to that sequence of consecutive imperative statements that must end in a period, an ELSE phrase associated with a previous IF statement, or a WHEN phrase associated with a previous SEARCH statement.

An imperative sentence is an imperative statement terminated by a period.

Arithmetic Expression

An arithmetic expression can be the identifier of a numeric elementary item, a numeric literal, a numeric function, such identifiers and literals separated by arithmetic operators, two arithmetic expressions separated by an arithmetic operator, or an arithmetic expression enclosed in parentheses.

A unary operator can precede any arithmetic expression. Table 6–1 shows the permissible combinations of variables, numeric literals, arithmetic operators, and parentheses.

Identifiers and literals appearing in arithmetic expressions must represent either numeric elementary items or numeric literals on which arithmetic can be performed. Numeric literals cannot exceed 98 digits when used for arithmetic operations.

The ways in which operators, variables, and parentheses can be combined in an arithmetic expression also appear in Table 6–1.

Table 6–1. Combination of Symbols in Arithmetic Expressions

First Symbol	Second Symbol				
	Variable	* / ** + −	Unary + or −	(	)
Variable	-	P	-	-	P
* / ** + −	P	-	P	P	-
Unary + or −	P	-	-	P	-
(	P	-	P	P	-
)	-	P	-	-	P

Legend

P indicates a permissible pair of symbols.
indicates an invalid pair.
Variable indicates an identifier or literal.

Arithmetic Operators

There are five binary arithmetic operators and two unary arithmetic operators that can be used in arithmetic expressions. They are represented by specific characters. The minus character (−) must be preceded and followed by a separator. The add (+), multiply (*), divide (/), and exponentiation (**) operators do not require separators.

Binary Arithmetic Operators	Meaning
+	Addition
−	Subtraction
*	Multiplication
/	Division
**	Exponentiation

Unary Arithmetic Operators	Meaning
+	The effect of multiplication by the numeric literal +1
−	The effect of multiplication by numeric literal −1.

Formation and Evaluation of Arithmetic Expressions

Parentheses can be used in arithmetic expressions to specify the order in which elements are to be evaluated. Expressions within parentheses are evaluated first. Within nested parentheses, evaluation proceeds from the least inclusive set. When parentheses are not used or parenthesized expressions are at the same level of inclusiveness, the following hierarchical order of execution is implied:

1. Unary plus (+) and minus (−)
2. Exponentiation (**)
3. Multiplication (*) and division (/)
4. Addition (+) and subtraction (−)

Parentheses are used either to eliminate ambiguities in logic, where consecutive operations of the same hierarchical level appear, or to modify the normal hierarchical sequence of execution in expressions when deviation is necessary. When parentheses do not specify the sequence of execution, operations of the same hierarchical level are executed from left to right.

For example the expression *A+B/C+D**E*F−G* would be interpreted as the following:

((A+(B/C)) + ((D**E) * F))−G

The sequence of operations works outward from the innermost parentheses; that is, first the exponentiation is performed *(D**E)*, then the multiplication of *(D**E)* by *F*, followed by division *(B/C)*. Addition and subtraction are last, proceeding from left to right; *A* is first added to *(B/C)*. This sum is then added to the next group, and *G* is subtracted.

An arithmetic expression can only begin with a left parenthesis, plus sign, minus sign, or a variable and can only end with a right parenthesis or a variable. There must be a one-to-one correspondence between left and right parentheses of an arithmetic expression so that each left parenthesis is to the left of its corresponding right parenthesis.

The following rules apply to evaluation of exponentiation in an arithmetic expression:

- If the value of an expression to be raised to a power is zero, the value of the exponent must not be greater than zero. Otherwise, the size error condition exists (see "SIZE ERROR Phrase" later in this section).
- If the evaluation yields both a positive and a negative real number, the result is the positive number.
- If no real number results from the evaluation, the size error condition exists.

When no resultant-identifier is associated with an expression, an intermediate data item is used to store the value of the arithmetic expression.

Intermediate Data Item

An intermediate data item is a signed numeric data item. This data item contains the values developed in the course of evaluating an arithmetic expression prior to the final value being moved to the resultant-identifier, if any.

An intermediate statement occurs when an arithmetic statement involves several operations. For example, *COMPUTE A = A * B + C* requires an intermediate item to contain the value of *A * B*; then, *C* is added to this intermediate item to produce the final result.

Caution

The length of the intermediate data item is limited to 98 decimal digits. It contains the leading zeroes, if any, and the leftmost digits of the value produced in the arithmetic operation. If at any time the number of significant digits generated exceeds 98 digits, the value is truncated on the right. This truncated value is used in the remainder of the computation.

For addition and subtraction operations, if the two operands, when aligned on their decimal points, require a field longer than the one provided for an intermediate data item, then one operand is truncated on the right end before the operation is performed.

For additional information refer to "COMPUTE Statement" in this section.

Conditional Expressions

Conditional expressions identify conditions that are tested to enable the object program to select between alternate paths of control depending upon the truth value of the condition. Conditional expressions are specified in the IF, PERFORM, and SEARCH statements. There are two categories of conditions associated with conditional expressions: simple conditions and complex conditions. Each can be enclosed within any number of paired parentheses, in which case its category is not changed.

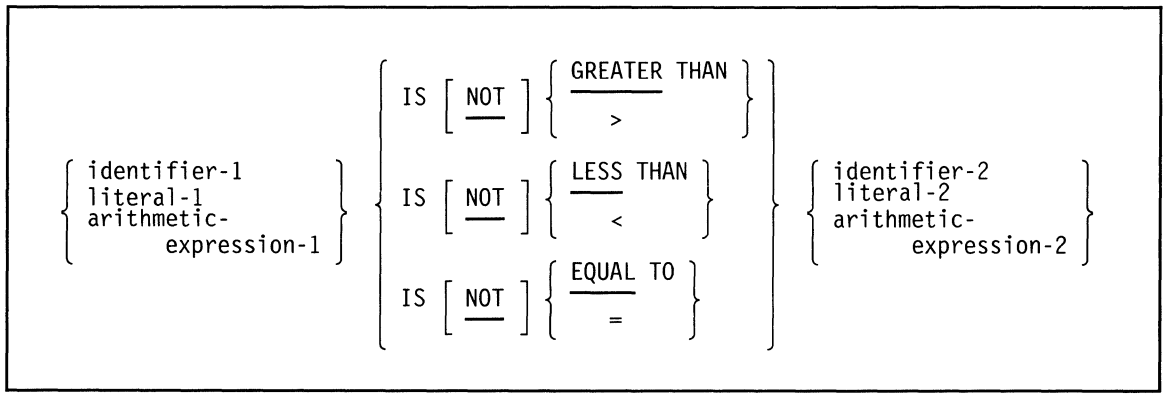
Simple Condition

The simple conditions are the relation, class, condition-name, switch status, and sign conditions. A simple condition has a truth value of true or false. The inclusion in parentheses of simple conditions does not change the simple truth value.

Relation Condition

A relation condition causes a comparison of two operands, each of which can be the data item referenced by an identifier, a literal, or the value resulting from an arithmetic expression. A relation condition has a truth value of true if the relation exists between the operands. Comparison of two numeric operands is permitted regardless of the formats specified in their respective USAGE clauses. However, for all other comparisons the operands must have the same usage. If either of the operands is a group item, the nonnumeric comparison rules apply.

The following is the general format:


identifier-1
literal-1
arithmetic-expression-1
IS [NOT] { GREATER THAN | > }
IS [NOT] { LESS THAN | < }
IS [NOT] { EQUAL TO | = }
identifier-2
literal-2
arithmetic-expression-2

Note: *The required relational characters ">", "<", and "=" are not underlined to avoid confusion with other symbols such as "≥" (greater than or equal to).*

The first operand (identifier-1, literal-1, or arithmetic-expression-1) is called the subject of the condition; the second operand (identifier-2, literal-2, or arithmetic-expression-2) is called the object of the condition. The relation condition must contain at least one reference to a variable.

The relational operator specifies the type of comparison to be made in a relation condition. A space, comma, semicolon, right parenthesis, closing quotation mark or closing commercial at sign (@) must precede the first reserved word comprising the relational operator. A space, comma, semicolon, left parenthesis, opening quotation mark or opening commercial at sign must follow the last reserved word comprising the relational operator. If the relational operator consists of more than one reserved word, then a space, comma, or semicolon must be used to separate each pair of consecutive reserved words comprising the relational operator. When used, NOT and the next keyword or relational character comprise one relational operator that defines the comparison to be executed for truth value; for example, NOT EQUAL is a truth test for an unequal comparison; NOT GREATER is a truth test for an equal or less comparison.

Comparison of Numeric Operands

For operands whose class is numeric (see "Class Condition" in this section), the algebraic value of the operands is compared. The length of the literal or arithmetic expression operands, in terms of number of digits represented, is not significant. Zero is considered a unique value regardless of the sign.

You can compare these operands regardless of the manner in which their usage is described. Unsigned numeric operands are considered positive for purposes of comparison.

Comparison of Nonnumeric Operands

For nonnumeric operands or for one numeric and one nonnumeric operand, a comparison is made with respect to a specified collating sequence of characters (see "Object-Computer Paragraph" in Section 4). If one of the operands is specified as numeric, it must be an integer data item or an integer literal and must meet the following requirements:

- If the nonnumeric operand is an elementary data item or a nonnumeric literal, the numeric operand is treated as though it had been moved to an elementary alphanumeric data item of the same size as the numeric data item. The contents of this alphanumeric data item are then compared to the nonnumeric operand (see "MOVE Statement" in this section and "PICTURE Clause", character "P", in Section 5).
- If the nonnumeric operand is a group item, the numeric operand is treated as though it were moved to a group item of the same size as the numeric data item. The contents of this group item are then compared to the nonnumeric operand (see "MOVE Statement" in this section and "PICTURE Clause", character "P" in Section 5.)
- A noninteger numeric operand cannot be compared to a nonnumeric operand.

The size of an operand is the total number of standard data format characters in the operand. Numeric and nonnumeric operands can be compared only when their usage is the same.

There are two cases to consider: operands of equal size and operands of unequal size.

- Operands of equal size. If the operands are of equal size, comparison proceeds by comparing characters in corresponding character positions, starting from the high order end and continuing until either a pair of unequal characters is encountered or the low order end of the operand is reached, whichever comes first. The operands are equal if all pairs of characters compare equally when the low order end is reached.

  The first encountered pair of unequal characters is compared to determine their relative position in the collating sequence. The operand that contains the character positioned higher in the collating sequence is considered to be the greater operand.
- Operands of unequal size. If the operands are of unequal size, comparison proceeds as though the shorter operand were extended on the right by sufficient spaces to make the operands of equal size.

Comparisons Involving Index-Names and/or Index Data Items

Relation tests can be made between the following:

- Two index-names. The result is the same as it would be if the corresponding ocurrence numbers were compared.
- An index-name and a data-item (other than an index data item) or literal. The occurrence number that corresponds to the value of the index name is compared to the data item or literal.
- An index data item and an index-name or another index data item. The actual values are compared without conversion.

The comparison of an index data item with any item or literal not previously specified is not allowed.

Comparisons Involving Mnemonic File Attributes

Mnemonic file attributes can appear as the subject of a mnemonic attribute relation condition; that is, a valid mnemonic-attribute-value is associated with the object of a relation condition in which a mnemonic-attribute-identifier appears as the subject.

Mnemonic-attribute relation conditions are allowed in any conditional expression. The following is the general format of a mnemonic attribute relation condition:

```
mnemonic-attribute-identifier IS [NOT] { EQUAL TO } mnemonic-attribute-value
                                       {    =    }
```

Mnemonic attribute relation conditions cannot be abbreviated.

For more information on file attributes, see Section 7 "File Attributes".

Class Condition

The class condition determines whether the operand is numeric (consisting entirely of the characters 0 through 9 with or without the operational sign) or alphabetic (consisting entirely of the characters A through Z and space). The following is the general format:

```
identifier IS [NOT] { NUMERIC    }
                    { ALPHABETIC }
```

The usage of the operand used with the ALPHABETIC test must be DISPLAY. The usage of the operand used with the NUMERIC test must be DISPLAY or COMPUTATIONAL. When used, NOT and the next keyword specify one class condition that defines the class test to be executed for truth value; for example, NOT NUMERIC is a truth test for determining that an operand is nonnumeric.

The NUMERIC test cannot be used with an item whose data description describes the item either as alphabetic or as a group item composed of elementary items whose data description indicates the presence of operational sign(s). If the data description of the item being tested does not indicate the presence of an operational sign, the item being tested is determined to be numeric only if the contents are numeric. If the data description of the item does indicate the presence of an operational sign, the item being tested is determined to be numeric only if the contents are numeric and a valid operational sign is present. The position and representation of valid operational signs are discussed under "PICTURE Clause" and "SIGN Clause" in Section 5.

The ALPHABETIC test cannot be used with an item whose data description describes the item as numeric. The item being tested is determined to be alphabetic only if the contents consist of any combination of the alphabetic characters A through Z and space.

Condition-Name Condition

In a condition-name condition, a conditional variable is tested to determine whether or not its value is equal to one of the values associated with a condition-name.

The following is the general format:

```
[NOT]  condition-name
```

If the condition-name is associated with a range or ranges of values, then the conditional variable is tested to determine whether or not its value falls in this range, including the end values. The result of the test is true if one of the values corresponding to the condition-name equals the value of its associated conditional variable. The rules for comparing a conditional variable with a condition-name value are the same as those specified for relation conditions.

Switch-Status Condition

A switch-status condition determines the ON or OFF status of a switch. The implementor-name and the ON or OFF value associated with the condition must be named in the SPECIAL-NAMES paragraph of the Environment Division.

The following is the general format:

```
[NOT]  condition-name
```

The result of the test is true if the switch is set to the position corresponding to the condition-name.

Sign Condition

The sign condition determines whether or not the algebraic value of an arithmetic expression is less than, greater than, or equal to zero.

The following is the general format:

```
                                        { POSITIVE }
arithmetic-expression IS [NOT]          { NEGATIVE }
                                        { ZERO     }
```

When used, NOT and the next keyword specify one sign condition that defines the algebraic test to be executed for truth value; for example, NOT ZERO is a truth test for a nonzero (positive or negative) value. An operand is POSITIVE if its value is greater than zero, NEGATIVE if its value is less than zero, and ZERO if its value is equal to zero. The arithmetic expression must contain at least one reference to a variable.

Complex Conditions

Form a complex condition by either combining simple conditions, combined conditions, and/or complex conditions with logical connectors (logical operators AND and OR), or by negating these conditions with logical negation (the logical operator NOT). The truth value of a complex condition, whether parenthesized or not, is the truth value that results from the interaction of all the stated logical operators on the individual truth values of simple conditions or on the intermediate truth values of conditions logically connected or logically negated.

The following are the logical operators and their meanings:

AND	Logical conjunction	The truth value is true if both of the conjoined conditions are true; false if one or both of the conjoined conditions is false.
OR	Logical inclusive OR	The truth value is true if one or both of the included conditions are true; false if both included conditions are false.
NOT	Logical negation or reversal of truth value	The truth value is true if the condition is false and false if the condition is true.

The logical operators must be preceded by a space and followed by a space. The values are summarized in Table 6–2.

Table 6–2. Truth Table for Logical Operators

Values of Condition		Values of Complex Condition		
A	**B**	**A AND B**	**A OR B**	**NOT A**
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

Negated Simple Conditions

A simple condition is negated through the use of the logical operator NOT, producing the opposite truth value. Thus the truth value of a negated simple condition is true if and only if the truth value of the simple condition is false; the truth value of a negated simple condition is false if and only if the truth value of the simple condition is true. The inclusion in parentheses of a negated simple condition does not change the truth value.

The following is the general format:

```
NOT simple-condition
```

Combined and Negated Combined Conditions

A combined condition results from connecting conditions with one of the logical operators AND or OR.

The following is the general format:

```
condition { { AND } condition } ...
          { { OR  }           }
```

In this format, condition can be any one of the following:

- A simple condition
- A negated simple condition
- A combined condition
- A negated combined condition; that is, the NOT logical operator followed by a combined condition enclosed within parentheses
- Combinations of the above four, specified according to the rules summarized in Table 6–3

Although parentheses need never be used when either AND or OR (but not both) is used exclusively in a combined condition, parentheses can be used to effect a final truth value when a mixture of AND, OR and NOT is used (see Table 6–3 and "Condition Evaluation Rules" in this section).

Table 6–3 indicates the ways in which conditions and logical operators can be combined and parenthesized. Left and right parentheses must have a one-to-one correspondence such that each left parenthesis is to the left of its corresponding right parenthesis.

In using Table 6–3, note, for example, that OR NOT is permissible, but the pair NOT OR is not; NOT followed by the left parenthesis is permissible, but NOT NOT is not.

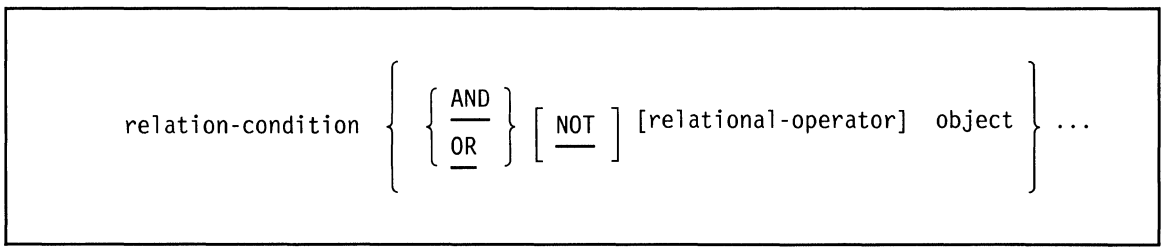
Table 6–3. Combinations of Conditions, Logical Operators, and Parentheses

Given the Following Element	Location in Conditional Expression		In a Left to Right Sequence of Elements:	
	First	Last	Element That May Immediately Precede Element When It Is Not First	Element That May Immediately Follow Element When It Is Not Last
Simple-condition	Yes	Yes	OR, NOT, AND, (	OR, AND,)
OR or AND	No	No	Simple-condition,)	Simple-condition, NOT, (
NOT	Yes	No	OR, AND, (	Simple-condition, (
(	Yes	No	OR, NOT, AND, (	Simple-condition, NOT, (
)	No	Yes	Simple-condition,)	OR, AND,)

Abbreviated Combined Relation Conditions

You can abbreviate any simple or negated simple relation condition other than the first one that appears in a combined conditional statement.

The following is the general format:



Within a sequence of relation conditions, both of the above forms of abbreviation can be used. The effect of using such abbreviations is the same as the effect if the last preceding stated subject were inserted in place of the omitted subject, and the last stated relational operator were inserted in place of the omitted relational operator. The result of such implied insertion must comply with the rules that are summarized in Table 6–4. This insertion of an omitted subject and/or relational operator terminates once a complete simple condition is encountered within a complex condition.

- You can omit the subject or the subject and relational operator. In these cases, the effect of the abbreviated relation condition is the same as it would be if the omitted parts had been taken from the nearest preceding complete relation condition within the same condition. That is, the first relation in a condition must be complete.
- You can abbreviate the sequence, if in a consecutive sequence of relation conditions without parentheses (separated by logical operators) the subjects, the relational operators, and the logical connectors are identical, as follows:
  - Abbreviation 1 - Identical subjects omitted in a consecutive sequence of relation conditions. An example of abbreviation 1 is:

    IF A = B AND = C

    This is equivalent to IF A = B AND A = C.
  - Abbreviation 2 - Identical subjects and relational operators omitted in a consecutive sequence of relation conditions. An example of Abbreviation 2 is

    IF A = B AND C

    This syntax is equivalent to IF A = B AND A = C.

The interpretation applied to the use of the word NOT in an abbreviated combined relation condition is as follows:

- If the word or symbol immediately following NOT is GREATER, >, LESS, <, EQUAL, or =, then NOT participates as a part of the relational operator
- Otherwise, the NOT is interpreted as a logical operator, and the implied insertion of subject or relational operator results in a negated relation condition.

The following are some examples of abbreviated combined and negated combined relation conditions and expanded equivalents:

Abbreviated Combined Relation Condition	Expanded Equivalent
a > b AND NOT < c OR d	((a > b AND (a NOT < c)) OR (a NOT < d))
a NOT EQUAL b OR c	(a NOT EQUAL b) OR (a NOT EQUAL c)
NOT a = b OR c	(NOT (a = b)) OR (a = c)
NOT (a GREATER b OR < c)	NOT ((a GREATER b) OR (a < c))
NOT (a NOT > b AND c AND NOT d)	NOT ((((a NOT > b) AND (a NOT> c)) AND (NOT (a NOT > d))))

Condition Evaluation Rules

Use parentheses to specify the order in which individual conditions of complex conditions are to be evaluated when the implied evaluation precedence cannot be used. Conditions within parentheses are evaluated first; within nested parentheses, evaluation proceeds from the least inclusive condition to the most inclusive condition. When you do not use parentheses or when parenthesized conditions are at the same level of inclusiveness, the following hierarchical order of logical evaluation is implied until the final truth value is determined:

1. Values are established for arithmetic expressions.
2. Truth values for simple conditions are established in the following order:
   a. Relation (following the expansion of any abbreviated relation condition)
   b. Class
   c. Condition-name
   d. Switch-status
   e. Sign
3. Truth values for negated simple conditions are established.
4. Truth values for combined conditions are established: AND logical operators are evaluated before OR logical operators.
5. Truth values for negated combined conditions are established.
6. The evaluation order for consecutive operations of the same hierarchical

level is from left to right when the sequence of evaluation is not completely specified by parentheses.

Following this logic, consider the following example.

```
IF CURRENT-MONTH AND DAY = 15 OR 30...
```

Evaluation for this example proceeds in the following order:

1. Truth value for the relation DAY = 15
2. Truth value of the simple condition-name condition (CURRENT-MONTH is true)
3. Truth value for combined conditions using AND (CURRENT-MONTH is TRUE AND DAY = 15)
4. Truth value for combined conditions using OR (the combined condition is the abbreviated combined relational, subject is DAY, object is = 30, so DAY = 30)

Thus, unless you use parentheses, there are two possible conditions that satisfy this example:

- CURRENT-MONTH is TRUE AND DAY = 15.
- DAY =30.

Numeric Functions: Offset Function

You can specify a numeric function as a sending operand in an arithmetic statement, a MOVE statement, or an arithmetic expression. The OFFSET function is a numeric function.

The general format of an OFFSET function is:

```
OFFSET ( data-name )
```

The OFFSET function represents the number of characters preceding the data item referenced by data-name in the logical record defining the data-name. The following rules apply to data-name in the OFFSET syntax:

- If data-name references a packed numeric data item not aligned on a character boundary, then the OFFSET (data-name) represents the number of characters preceding the character in which data-name begins.
- If data-name is a record-name or a 77-level item, the value of the OFFSET function is ZERO.

Data-name can be qualified. The internal representation of OFFSET is the same as that of an index data item.

Common Phrases

In the statement descriptions that follow, several phrases appear frequently: the ROUNDED phrase, the SIZE ERROR phrase, and the CORRESPONDING phrase.

In the following discussion, a resultant-identifier is that identifier associated with the result of an arithmetic operation.

ROUNDED Phrase

If, after decimal point alignment, the number of places in the fraction of the result of an arithmetic operation is greater than the number of places provided for the fraction of the resultant-identifier, truncation is relative to the size provided for the resultant-identifier. When you request rounding, the absolute value of the resultant-identifier is increased by adding a one into its low-order digit whenever the absolute value of the next least significant digit of the intermediate data item is greater than or equal to five.

When the character *P* represents the low-order integer positions in a resultant-identifier in the picture for that resultant-identifier, rounding or truncation occurs relative to the rightmost integer position for which storage is allocated.

SIZE ERROR Phrase

If, after decimal point alignment, the absolute value of a result exceeds the largest value that can be contained in the associated resultant-identifier, a size error condition exists. Division by zero always causes a size error condition.

Note: *The size error condition applies only to the final results of an arithmetic operation, not to intermediate results, except in the MULTIPLY and DIVIDE statements. Consequently an intermediate result of a COMPUTE statement could exceed the 98-digit length limit of the Intermediate Data Item. However, a size error condition would not result unless the final results of the COMPUTE exceed the limit of the resultant-identifier. Such a condition could produce unexpected results.*

If you specify the ROUNDED phrase, rounding occurs before size error is checked. When there is a size error, the subsequent action depends on whether or not the SIZE ERROR phrase is specified.

- If the SIZE ERROR phrase is not specified and a size error condition occurs, the value of the resultant-identifier(s) contains the final resultant value moved in with left truncation.
- If the SIZE ERROR phrase is not specified and a size error condition occurs when the $OPTIMIZE dollar-option is set, the final resultant value is not moved to the resultant-identifier or identifiers.
- If the SIZE ERROR phrase is specified and a size error condition occurs, then the values of resultant-identifier or identifiers affected by the size errors are not altered. Values of resultant-identifier or identifiers for which no size error condition occurs are unaffected by size errors that occur for the other resultant-identifier or identifiers. After completion of this operation, the imperative statement in the SIZE ERROR phrase is executed.
- If any of the individual operations produces a size error condition, for the ADD statement with the CORRESPONDING phrase and the SUBTRACT statement with the CORRESPONDING phrase, the imperative statement in the SIZE ERROR phrase is not executed until all of the individual additions and subtractions are completed.

CORRESPONDING Phrase

CORR is an abbreviation for CORRESPONDING.

For the purpose of this discussion, d1 and d2 must be identifiers that refer to group items. A pair of data items, one from d1, and one from d2, correspond if the following conditions exist:

- A data item in d1 and a data item in d2 are not designated by the keyword FILLER and have the same data-name and the same qualifiers up to, but not including, d1 and d2.
- At least one of the data items is an elementary data item in the case of a MOVE statement with the CORRESPONDING phrase, and both of the data items are elementary numeric data items in the case of either the ADD statement with the CORRESPONDING phrase or the SUBTRACT statement with the CORRESPONDING phrase.
- The description of d1 and d2 must not contain level number 66, 77, or 88 or the USAGE IS INDEX clause.
- A data item that is subordinate to d1 or d2 and contains a REDEFINES, RENAMES, OCCURS, or USAGE IS INDEX clause is ignored, as well as those data items subordinate to the data item that contains the REDEFINES, OCCURS, or USAGE IS INDEX clause. However, d1 and d2 can have REDEFINES or OCCURS clauses or be subordinate to data items with REDEFINES or OCCURS clauses.

General Rules for Statement Formats

The following paragraphs describe some general rules for statement formats.

Arithmetic Statements

The arithmetic statements are the ADD, COMPUTE, DIVIDE, MULTIPLY, and SUBTRACT statements. They have the following common features:

- The data descriptions of the operands need not be the same; any necessary conversion and decimal point alignment is supplied throughout the calculation.
- The maximum size of each operand is 98 decimal digits.
- Each arithmetic operation is evaluated by the use of an intermediate data item for the result of the operation. If the size of the result being developed is greater than the size of the intermediate data item, the result is truncated on the right to the size of the intermediate data item (refer to "Intermediate Data Item" in this section.) The contents of the intermediate data item are moved to the resultant-identifier according to the rules for the MOVE statement. Rounding is performed, and the size error condition is determined only during this move.

Overlapping Operands

When a sending and a receiving item in an arithmetic statement or an INSPECT, MOVE, SET, STRING, or UNSTRING statement share a part of their storage areas, the result of executing such a statement is undefined.

The overlapping operand rule applies only to operands that share a part but not all of their storage areas. Overlapping moves are allowed.

Multiple Results in Arithmetic Statements

The ADD, COMPUTE, DIVIDE, MULTIPLY, and SUBTRACT statements can have multiple results. Such statements behave as though they had been written in the following way:

- A statement that performs all arithmetic necessary to get the result to be stored in the receiving items and then stores that result in a temporary storage location.
- A sequence of statements transferring or combining the value of this temporary location with a single result. These statements are written in the same left-to-right sequence that the multiple results are listed:

  The result of the statement *ADD a, b, c TO c, d (c), e* is equivalent to the following:

```
ADD a, b, c GIVING temp
ADD temp to c
ADD temp to d (c)
ADD temp to e
```

In these statements, *temp* is an intermediate data item provided by the compiler. Refer to "Intermediate Data Item" in this section.

Incompatible Data

Except for the class condition (see "Simple Condition" and "Class Condition" in this section), when the contents of a data item are referenced in the Procedure Division and the contents of that data item are not compatible with the class specified for that data item by its PICTURE clause, the result of such a reference is undefined.

Specific Formats

The following are specific statement formats with a detailed discussion of the restrictions and limitations associated with each. The verbs appear in alphabetical order.

ACCEPT Statement

he ACCEPT statement makes low volume data available to a specified data item.

The following are the general formats:

Format 1

```
ACCEPT identifier [ FROM {mnemonic-name} ]
```

Format 2

```
                                  { DATE          }
                                  { DAY           }
                                  { TIME          }
ACCEPT identifier FROM            { TIMER         }
                                  { TODAYS-DATE   }
                                  { TODAYS-NAME   }
```

Format 3

```
ACCEPT identifier-1 MESSAGE COUNT
```

Format 1

The ACCEPT statement transfers data from the hardware device. This data replaces contents of the data item named by the identifier.

The maximum number of characters that can be transferred is unlimited.

You must also specify mnemonic-name in Format 1 in the SPECIAL-NAMES paragraph of the Environment Division. Mnemonic-name must be associated with the hardware-name ODT.

The number of characters tranferred with the ACCEPT statement must correspond to the size of the receiving data-name. If you enter fewer characters, an ETX character followed by undefined data is in the message area. If you enter more characters, the excess characters are truncated.

Format 2

The ACCEPT statement tranfers the information requested to the data item specified by the identifier according to the rules of the MOVE statement.

DATE, DAY, TIME, TIMER, TODAYS-DATE, and TODAYS-NAME are conceptual data items and, therefore, are not described in the COBOL program. These fields appear as follows.

DATE

This special register is composed of the following data elements: year of century, month of year, and day of month. The sequence of the data element codes is from high order to low order (left to right): year of century, month of year, and day of month. Therefore, July 1, 1990 is expressed as 900701. DATE, when accessed by a COBOL program, behaves as if it had been described in the COBOL program as an unsigned elementary numeric integer data item six digits in length (PIC 9(6) COMP).

DAY

This special register is composed of the following data elements: year of century and day of year. The sequence of the data element codes is from high order to low order (left to right) year of century, day of year. Therefore, July 1, 1990 is expressed as 90183. DAY, when accessed by a COBOL program, behaves as if it had been described in a COBOL program as an unsigned elementary numeric integer data item five digits in length (PIC 9(5) COMP).

TIME

This special register is composed of the following data elements: hours, minutes, seconds and hundredths of a second. TIME is based on elapsed time after midnight on a 24-hour clock basis. Thus, 2:41 p.m. is expressed as 14410000. TIME, when accessed by a COBOL program, behaves as if it had been described in COBOL as an unsigned elementary numeric integer data item eight digits in length (PIC 9(8) COMP). The minimum value of TIME is 00000000; the maximum value of TIME is 23595999. If the hardware does not have the facility to provide fractional parts of TIME, the value is converted to the closest decimal approximation.

TIMER

This special register is composed of the current value of the object computer's interval timer (that is, the current system time in milliseconds). TIMER, when accessed by a COBOL program, behaves as if it had been described in COBOL

as an unsigned elementary numeric integer data item ten digits in length (PIC 9(10) COMP).

TODAYS-DATE

This special register is composed of the following data elements: month of year, day of month, and year of century. Therefore, July 1, 1990 would be expressed as 070190. TODAYS-DATE, when accessed by a COBOL program, behaves as if it had been described in COBOL as an unsigned six digit elementary numeric integer data item (PIC 9(6) COMP).

TODAYS-NAME

This special register is composed of the name of the current day of the week. When accessed by a COBOL program, TODAYS-NAME behaves as if it had been described in COBOL as an elementary data item, nine alphanumeric characters in length (PIC X(9)), with the name of the current day left justified with space fill as its value.

Format 3

The ACCEPT MESSAGE COUNT statement returns a count of the number of entries in a storage queue into the entry-data-length field of the specified STOQUE parameter block.

The structure of a STOQUE parameter block is as follows:

```
01  Identifier-1.
    02  Queue-name             PIC X(6).
    02  Entry-name-length      PIC 9(2) COMP.
    02  Entry-name             PIC X(nn).
    02  Entry-data-length      PIC 9(4) COMP.
    02  Entry-data             PIC X(nnnn).
```

See "SEND Statement" later in this section for a complete description of the storage queue function and the STOQUE parameter block.

Identifier-1 is the name of a STOQUE parameter block. Its queue-name field must contain the name of the queue to be interrogated. An entry-name can be given in the STOQUE parameter block to specify a particular element or group of elements in the queue (see "SEND Statement" and "RECEIVE Statement" in this section). In this case, the count returned is the number of entries in the queue for that entry-name or name-group. If entry-name is not specified, the count returned is the total number of entries in the queue. The response is returned as an unsigned integer in the entry-data-length field of the specified STOQUE parameter block. A response of zero means that the queue or the designated portion of the queue is empty.

The MCPs STOQ option must be set to use this statement.

ADD Statement

The ADD statement causes two or more numeric operands to be summed and the result to be stored.

The following are the general formats:

Format 1

```
ADD { identifier-1 } [ ,identifier-2 ]  ... TO identifier-m [ ROUNDED ]
    { literal-1    } [ ,literal-2    ]

    [ ,identifier-n [ROUNDED] ] ... [ ; ON SIZE ERROR imperative-statement ]
```

Format 2

```
ADD { identifier-1 } , { identifier-2 } [ , identifier-3 ]  ...
    { literal-1    }   { literal-2    } [ , literal-3    ]

    GIVING identifier-m [ROUNDED] [ ,identifier-n [ROUNDED] ] ...

    [ ; ON SIZE ERROR imperative-statement ]
```

Format 3

```
ADD { CORRESPONDING } identifier-1 TO identifier-2 [ ROUNDED ]
    { CORR          }

    [ ; ON SIZE ERROR imperative-statement ]
```

If Format 1 is used, the values of the operands preceding the word TO are added together. Then the sum is added to the current value of identifier-m, and the result is stored immediately into identifier-m. This process is repeated respectively for each operand following the word TO.

If Format 2 is used, the values of the operands preceding the word GIVING are added together. Then the sum is stored as the new value of each identifier-m and identifier-n (the resultant-identifiers).

If Format 3 is used, data items in identifier-1 are added to and stored in corresponding data items in identifier-2. Intermediate results are limited to 98 digits, which is the size of the intermediate data item (see "Intermediate Data Item" in this section).

In Formats 1 and 2, each identifier must refer to an elementary numeric item, except that in Format 2 each identifier following the word GIVING must refer to either an elementary numeric item or an elementary numeric-edited item. In Format 3, each identifier must refer to a group item.

Each literal must be a numeric literal.

Additional rules and explanations relative to this statement are given in the appropriate paragraphs. (See "Intermediate Data Item", "ROUNDED Phrase", "SIZE ERROR Phrase", "CORRESPONDING Phrase", "Arithmetic Statements", "Overlapping Operands", and "Multiple Results in Arithmetic Statements," earlier in this section.)

ALTER Statement

The ALTER statement modifies a predetermined sequence of operations.

The general format is:

```
ALTER procedure-name-1 TO [ PROCEED TO ] procedure-name-2

[ , procedure-name-3 TO [ PROCEED TO ] procedure-name-4 ] ...
```

Each procedure-name-1 and procedure-name-3 is the name of a paragraph that contains a single sentence consisting of a GO TO statement without the DEPENDING phrase.

Each procedure-name-2 and procedure-name-4, the objects of the PROCEED TO clause, is the name of a paragraph or section in the Procedure Division.

Executing the ALTER statement modifies the GO TO statement in the paragraphs named procedure-name-1 and procedure-name-3, so that subsequent executions of the modified GO TO statements transfer control to procedure-name-2 and procedure-name-4, respectively. An ALTER statement can refer to a GO TO statement in a section whose segment-number ends in a number greater than 049 (that is, an independent segment). However, the segment number containing the ALTER statement must have the same segment number as that of the segment containing the GO TO.

All other uses of the ALTER statement are valid.

Procedure-name-1 and procedure-name-3 can appear in a section whose segment-number ends in a number greater than a specified SEGMENT-LIMIT and less than 050 (that is, a fixed overlayable segment).

Modified GO TO statements in fixed overlayable segments are always returned to their last used state when the segment is reloaded. Modified GO TO statements in INDEPENDENT overlayable segments return to their original state.

For complete information on independent and fixed overlayable segments, refer to "Segmentation" earlier in this section.

CALL Statement

The CALL statement transfers control from one program to another during program execution. You can write called programs in COBOL ANSI-74, FORTRAN ANSI-77, Pascal, or BPL.

The following is the general format:

```
CALL    { identifier-1 }  [ DISKPACK  pack-name ]
        { literal-1    }  [ DISK                ]
        [ USING data-name-1 [ , data-name-2 ] ... ]
        [ ; ON OVERFLOW imperative-statement  ]
```

Define identifier-1 as an alphanumeric data item with a value consistent with program-naming conventions (that is, 6-character filenames) and literal-1 as a nonnumeric literal. Literal-1 or identifier-1 are names of called programs.

Interlanguage binding is permitted without a stand-alone binder when you are calling non-COBOL programs by specifying the following syntax for literal-1:

```
"<icm-name>.<program-name>.<entry-point>"
```

This syntax must be enclosed in quotation marks. Periods are required between the variables.

Icm-name is the name of a BPL, Pascal, FORT77, or COBOL ICM that has been created.

Program-name is a module name in BPL, program name in Pascal, subroutine name in FORT77, or program-id in COBOL.

Entry-point is a procedure name in BPL and Pascal, subroutine name in FORT77, or PROG_FIRST_EXECUTABLE in COBOL.

Note: *If you use the literal-1 option, neither the called program nor the program containing the CALL statement (the calling program) should be compiled with the $LEV2IPC dollar-option set.*

If you use the literal-1 option and the called program is not a COBOL program, the following considerations apply:

- The called program must be compiled with $ICM dollar-option set.
- The called program must have been compiled, and the resulting ICM must be present on the system, before the calling program is compiled.

At run time identifer-1 should contain the name of the called program.

Note: *If you use the identifier-1 option, both the calling program and the called program must be compiled with the $LEV2IPC compiler control option set. All programs must be compiled with the same dollar-option.*

The pack-name is the diskpack on which the called program resides. In level 1 IPC, the pack-name must be a literal. When you specify $LEV2IPC, the pack-name can be either a literal or an identifier.

Note: *If you are specifying a particular pack destination for level 1 IPC, the called program must be compiled with the $ICM dollar option set. If you use the multi-file-id and device PACK options, the called program is created on the destination specified. The compiler ignores the file-id, and uses instead the first six characters of the PROGRAM-ID paragraph.*

The following rules designate the location of the called program or module:

- When you specify the ICMFAMILY dollar option and omit the DISKPACK or DISK clause, the called program is assumed to reside on the 180-byte diskpack media specified in the ICMFAMILY dollar option.
- When you specify neither the ICMFAMILY dollar option nor the DISKPACK/DISK clause, the called module is assumed to reside on 100-byte disk media.
- If you specify both the ICMFAMILY dollar option and the DISK/DISKPACK clause, the called program is assumed to reside on the media specified in the DISK/DISKPACK clause.

You can use the optional USING phrase only if you included a USING phrase in the Procedure Division statement of the called program.

Note: *The number of operands in each USING phrase (CALL and Procedure Division statements) must be identical.*

The data-names specified in the USING phrase indicate the data items that can be referred to in the called program. They are therefore available to the calling program. The order in which you include data-names in the USING phrase of the CALL statement and Procedure Division statement determines data correspondence. For example, the data item relative to data-name-1 in the CALL statement corresponds to the data item relative to data-name-1 in the Procedure Division statement. Note that data correspondence is relative to position, not name; however, there is no correspondence with index-names.

You must define each of the operands in the USING phrase as a data item in the File, Working-Storage, or Linkage sections of the program. The operand also must be a level number 01 or 77. Additionally, you can define operands in the USING phrase as file-names, as specified in the SELECT clause of the Environment Division.

When a file-name is specified in an interprogram communication (IPC) call, an 8 digit FIB address is passed to the called program. Data-name-1, data-name-2, and so forth can be qualified when they reference items defined in the File Section.

The following are the functions of the ON OVERFLOW statement:

- If the called program is not present on disk, the ON OVERFLOW imperative statement is executed and the CALL is cancelled.
- If the called program is not present on disk and the ON OVERFLOW option is not present, the calling program terminates.
- If the called program is present, the ON OVERFLOW option is ignored.

A called program is in an initial state the first time it is called either during a program execution or after a CANCEL (refer to "CANCEL Statement" later in this section). All other entries (data entries, status and positioning of files, and all alterable switch settings) do not change the program status. Called programs can contain CALL statements; however, these statements cannot directly or indirectly call the calling program.

The CALL statement can appear anywhere within a segmented program. When a CALL statement appears in a section with a segment-number greater than or equal to 50, that segment is in its last used state when the EXIT PROGRAM statement returns control to the calling program.

CALL INTRINSIC Statement

The CALL INTRINSIC statement provides System Dependent Intrinsics (nonstandard, system-dependent features) and SHARED file functions for the COBOL programmer.

The following are the general formats for System Dependent Intrinsics (SDIs):

Format 1

```
CALL "INTRINSIC ABORT"
```

Format 2

```
CALL "INTRINSIC { DATE-COMPILED }
                { DATE-NOW      }
                { MIX           } " USING identifier-1
                { TIME-NOW      }
                { SETSWITCH     }
```

Format 3

```
CALL "INTRINSIC { GETPARAM          }
                { RESULT-DESCRIPTOR }
                { GETSWITCH         } " GIVING identifier-1
                { MIXTBL            }
                { PROGINFO          }
                { JOBINFO           }
```

Format 4

```
CALL "INTRINSIC  { MIXID  }  " USING program-id GIVING identifier-1
                 { MIXNUM }
```

Formats 5 and 6 are the general formats for SHARED file functions:

Format 5

```
                 { SEEKLOCKONLY }
CALL "INTRINSIC  { SEEKLOCK     }  " USING file-name
                 { SEEKLOCKWAIT }
```

Format 6

```
                 { READLOCKONLY }
CALL "INTRINSIC  { READLOCKWAIT }  " USING file-name
                 { READLOCK     }

                 { WRITELOCK    }   " USING record-name

                 [ ON OVERFLOW  imperative-statement ]

CALL "INTRINSIC    UNLOCK"          USING record-name
```

Format 7

```
CALL "INTRINSIC DISKFILEHEADER"

    USING { identifier-1 } [ DISKPACK { identifier-2 } ]
          { literal-1    }             { literal-2    }

    GIVING identifier-3
```

Format 8

```
CALL "INTRINSIC { SPOMESSAGE } "
                { GETMCP     }

        USING { literal-1    }
              { identifier-1 }

        GIVING identifier-2
```

Format 9

```
CALL "INTRINSIC ZIPSPO"   USING { literal-1    }
                                { identifier-1 }

                          GIVING identifier-2
```

Format 10

```
CALL "INTRINSIC { BINARY-DECIMAL } "
                { DECIMAL-BINARY }

     USING      { literal-1    }    GIVING identifier-2.
                { identifier-1 }

CALL "INTRINSIC TRANSLATE"

     USING      { literal-3    }    , identifier-4
                { identifier-3 }

     GIVING   identifier-5.
```

Format 11

```
CALL "INTRINSIC GETLABEL"

     USING  file-name-1  GIVING identifier-1.

CALL "INTRINSIC PUTLABEL"

     USING file-name-2 , identifier-2
```

Format 1

CALL "INTRINSIC ABORT" terminates the calling program abnormally. This termination results from generating the invalid operation code (instruction) *EC* in the code stream.

Format 2

CALL "INTRINSIC DATE-COMPILED" obtains the compiling date and time of the calling program in the form hhmmMMDDYY, where hh = hour, mm = minute, MM = month, DD = day, and YY = year. The result is stored in identifier-1. Identifier-1 must be described in the COBOL program as an elementary numeric integer data item of at least 10 digits.

CALL "INTRINSIC DATE-NOW" obtains the symbolic representation of the current date in the form MMM DD YYYY or MMM D YYYY, where MMM is the abbreviated month-name, DD or D is the day of the month, and YYYY is the year. The result is stored in identifier-1. Identifier-1 must be described in the COBOL program as an unsigned elementary alphanumeric data item of at least 12 characters.

CALL "INTRINSIC MIX" obtains the number of programs in the mix. The number is placed in identifier-1. Identifier-1 must be described in the COBOL program as an elementary numeric integer data item of at least 4 digits.

CALL "INTRINSIC TIME-NOW" obtains the symbolic representation of the current time in the form HH:MM XXXX, where XXXX = A.M. or P.M. The result is stored in identifier-1. Identifier-1 must be described in the COBOL program as an unsigned elementary alphanumeric data item of at least 10 characters.

CALL "INTRINSIC SETSWITCH" causes the value of identifier-1 to be moved to the switch location. Identifier-1 must be defined as an 8-digit computational numeric data item.

Format 3

CALL "INTRINSIC GETPARAM" causes the values in the insert parameter to be moved to identifier-1. This identifier must be described in the program as a 12-character alphanumeric data item.

CALL "INTRINSIC RESULT-DESCRIPTOR" causes the 4-digit result descriptor to be moved to identifier-1. This identifier must be described in the program as a 4-digit computational numeric data item.

Note: *The program must be Armed (Refer to "USE Statement, Format 4" in this section) to successfully use the RESULT-DESCRIPTOR intrinsic.*

CALL "INTRINSIC GETSWITCH" causes the current values of the switches to be moved to identifier-1. A value can then be listed in any manner consistent with the identifier-1 description, an 8-digit computational numeric data item.

CALL "INTRINSIC MIXTBL" causes information from the MCP mix table to be returned to the calling program. The mix table contains information about the program name, multi-program name, memory space, and mix number of every job in the time-sharing mix. Identifier−1 must be a data item conforming to the following format:

```
02 IDENTIFIER-1.
   03  HEADER.
       05  JOBS-IN-MIX                  PIC 9(4)   COMP.
       05  MEM-AVAILABLE                PIC 9(10)  COMP.
   03  ENTRY OCCURS (nnn) TIMES.
       05  PROGRAM-NAME                 PIC X(6).
       05  MULTI-PROGRAM-NAME           PIC X(6).
       05  MIX-NUMBER                   PIC 9(4)   COMP.
       05  MEMORY-USED-BY-JOB           PIC 9(7)   COMP.
       05  PROCESSOR-PRIORITY           PIC 9      COMP.
       05  MEMORY-PRIORITY              PIC 9      COMP.
       05  SPECIAL-PROGRAM-CODE         PIC 9      COMP.
*         {1 = program is a generator
*          2 = program is DMPALL
*          5 = DSKOUT or pack SQUASH
*          6 = program has DCP or MCS status
*          7 = timesharing process
*          8 = timesharing handler
*          A = generator in shared area
*          B = DMS control program
*          C = WFL handler
*          D = BNA handler
*          E = program is copy}
       05  PROGRAM-STATUS-CODE          PIC 99     COMP.
*         {00 = executing
*          01 = compiler
*          02 = waiting IO
*          03 = waiting core-to-core
*          04 = stopped
*          05 = no complex wait table space
*          06 = waiting complex wait
*          07 = waiting stoque entry
*          08 = waiting stoque memory
*          09 = waiting stoque name slot
*          10 = waiting stoque processing
*          11 = waiting trace
*          12 = waiting operator action
*          14 = waiting memory
*          15 = sleeping}
```

Note: *The nnn refers to the number of entries that you allocate. It is your responsibility to allocate a table of sufficient size.*

CALL "INTRINSIC PROGINFO" causes the MCP to return to the program a structure containing information about the job running. Identifier-1 must be an alphanumeric item of at least 52 digits. The data returned in identifier-1 conforms to the following format:

```
02  IDENTIFIER-1.
    03  PROGRAM-NAME                  PIC X(6).
    03  MULTI-PROGRAM-NAME            PIC X(6).
    03  MIX-NUMBER                    PIC 9(4)   COMP.
```

```
    03  RLOG-NUMBER                    PIC 9(4)   COMP.
    03  RJE-LINK                       PIC 9(4)   COMP.
    03  DATE-COMPILED                  PIC 9(6)   COMP.
    03  TIME-COMPILED                  PIC X(5).
```

CALL "INTRINSIC JOBINFO" causes the MCP to return to the program a structure containing information about the job environment. Identifier-1 must be an alphanumeric data item of at least 200 digits. The data returned in identifier-1 conforms to the following format:

```
02  IDENTIFIER-1.
    03  RESERVED-1                     PIC X(2).
    03  MCP-NAME                       PIC X(17).
    03  MCP-RELEASED-LEVEL             PIC X(4).
    03  MCP-PATCH-LEVEL                PIC X(6).
    03  MCP-VERSION-DATE               PIC 9(6)   COMP.
    03  PROCESSOR-NUMBER-OF-CALLER     PIC 9(2)   COMP.
    03  HOSTNAME                       PIC X(17).
    03  MIX-NUMBER-OF-CALLER           PIC 9(4)   COMP.
    03  BATCH-OR-TIMESHARING-FLAG      PIC 9      COMP.
    03  RESERVED-2                     PIC 9(95)  COMP.
```

Format 4

CALL "INTRINSIC MIXID" obtains the number of programs in the mix with an identifier indicated by program-id. The program-id must be a 6-character alpha literal or an identifier of six characters. The number is placed in identifier-1. This identifier must be described in the COBOL program as an elementary data item of at least four digits.

CALL "INTRINSIC MIXNUM" obtains the mix number of the program identified by program-id. The program-id must be a 6-character alpha literal or an identifier of six characters. The mix number is placed in identifier-1. This identifier must be described in the COBOL program as an elementary numeric integer data item of at least four digits. If multiple programs with the same program-id are in the mix, the mix number returned is any one of them.

Format 5

CALL "INTRINSIC SEEKLOCKONLY" requests that a block of a shared disk or diskpack file be locked without any transfer of data. This operation is identical to READLOCKONLY except that the program does not wait if the block is currently LOCKed. The applicable file must be opened I/O at the time the call is executed.

CALL "INTRINSIC SEEKLOCK" initiates a SEEK for a specified record and, when the record is found, locks the block. The program continues to process. A subsequent READ is still necessary to make the record available. Seek operations permit some overlap of processing and I/O. The applicable file must be opened I/O when you execute the CALL.

CALL "INTRINSIC SEEKLOCKWAIT" initiates a SEEK for a specified record and, when the record is found, locks the block, transfers the information to the program's buffer, and then unlocks it. The program continues. A subsequent READ is still necessary to make the record available. The applicable file must be opened I/O when the CALL is executed.

SEEKLOCKWAIT is similiar to READLOCKWAIT, except that program processing can continue and the record is not made available in the record area.

Format 6

CALL "INTRINSIC READLOCKONLY" requests that a block of a shared disk or diskpack file be locked without any transfer of data. If the block is currently locked, the program waits to relock it. The applicable file must be opened I/O at the time the CALL is executed.

CALL "INTRINSIC READLOCKWAIT" locks a block of a shared file, performs a read, and then unlocks the block. The applicable file must be opened I/O at the time the CALL is executed.

If the requested block is locked by another program, READLOCKWAIT waits until the record is unlocked. If the block is already locked by the program issuing the READLOCKWAIT, no delay is necessary.

READLOCKWAIT obtains the contents of a record at a time when no other program has the record locked. READLOCKWAIT differs from READLOCK in that READLOCKWAIT does not leave the record locked.

CALL "INTRINSIC READLOCK" reads and locks a block of a file. The applicable file must be opened I/O at the time the CALL is executed.

CALL "INTRINSIC WRITELOCK" writes the current record of a shared disk or diskpack file, but does not unlock it after the write. Therefore, the record is not available to other programs. The record can be released either by the UNLOCK intrinsic or with a subsequent simple WRITE statement.

You cannot use WRITELOCK unless the current record has previously been locked. A program that attempts to carry out a WRITELOCK action on an unlocked record is terminated.

CALL "INTRINSIC UNLOCK" requests that a block of a shared disk or diskpack file be unlocked.

If the associated ACTUAL KEY is invalid, the imperative statement is executed unless a USE procedure is declared for the file.

The ON OVERFLOW phrase or an applicable USE statement must be specified for the functions READLOCK, READLOCKWAIT, READLOCKONLY, and WRITELOCK. The ON OVERFLOW condition exists and the imperative statement is executed if the content of the associated ACTUAL KEY data item is found to be invalid.

To indicate the action to be taken when a STALEMATE condition exists for shared files, you must do the following:

- Specify a FILE STATUS clause in the FILE-CONTROL section of the COBOL program
- Specify an applicable USE statement for handling SHARED file errors in the DECLARATIVES section of the COBOL program.
- Check for the timeout condition (value 96) on the STATUS key variable in the USE statement of the program. Indicate statements to be executed if a timeout has occurred.

If there is no USE statement declared for SHARED file error handling, a stalemate condition causes a program DS or DP.

Format 7

CALL "INTRINSIC DISKFILEHEADER" causes 40 digits of the disk file header to be returned to the requestor. Identifier-1 or literal-1 is the name of the file. When used, identifier-2 or literal-2 is the name of the diskpack where the file resides. Identifier-1, literal-1, identifier-2, and literal-2 must be 6-byte alphanumeric items, and must match the file name exactly.

The 40 digit results are returned in identifier-3. Therefore, identifier-3 must be defined as a 40-digit computational numeric data item. If the file indicated in identifier-1 or identifier-2 or literal-1 or literal-2 is not present, identifier-3 contains zeros. The data returned in identifier-3 conforms to the following format:

```
02  IDENTIFIER-3.
    03  RECORD-SIZE-IN-DIGITS              PIC 9(5)  COMP.
    03  RECORDS-PER-BLOCK                  PIC 9(3)  COMP.
    03  NUMBER-OF-AREAS                    PIC 9(2)  COMP.
    03  END-OF-FILE-POINTER                PIC 9(8)  COMP.
    03  NUMBER-OF-USERS-ON-PROCESSOR-0     PIC 9(2)  COMP.
    03  NUMBER-OF-USERS-ON-PROCESSOR-1     PIC 9(2)  COMP.
    03  NUMBER-OF-USERS-ON-PROCESSOR-2     PIC 9(2)  COMP.
    03  NUMBER-OF-USERS-ON-PROCESSOR-3     PIC 9(2)  COMP.
    03  RESERVED                           PIC 9(5)  COMP.
    03  FILE-TYPE-1                        PIC 9     COMP.
    03  FILE-TYPE-2                        PIC 9     COMP.
    03  SECTORS-PER-AREA                   PIC 9(7)  COMP.
```

Format 8

CALL "INTRINSIC SPOMESSAGE" causes keyboard input messages to be passed to the MCP and requests that the response be returned to the program.

The following are the only keyboard input messages that can be used with "INTRINSIC SPOMESSAGE".

AJ	BF	CA	CC	CD	CK
CN	DA	DB	DC	DQ	DX
FI	FM	FN	FR	GO	GT
HN	IL	IN	LC	LI	LP
MR	MX	NET	NL	NT	OF
OK	OL	OT	OU	PD	RA
RD	RF	RO	SB	SK	SO
SW	TI	TO	UP	UR	WB
WC	WD	WJ	WM	WO	WQ
WS	WT	WX	WY	XC	

The use of any other input message results in the following message from the MCP:

```
<BEL><BEL> ** KBD IGNORED: REQUEST NOT ALLOWED.
```

Certain MCP control instructions can also be passed with CALL "INTRINSIC SPOMESSAGE" including the following:

ALLOCATE	ALTER	CHANGE	DEBUG
DISK	DISPLAY	FILE	INSERT
PATCH	SHOW	SPO	TIME
USER	WHATS		

In the USING phrase of CALL "INTRINSIC SPOMESSAGE", literal-1 or identifier-1 must be a nonnumeric literal or an alphanumeric data item containing a valid keyboard input message or control instruction. Literal-1 or identifier-1 must be in the same format as that of any ODT message typed at the ODT and should end with a period or an ETX character.

In the GIVING phrase of CALL "INTRINSIC SPOMESSAGE", each response line is returned in identifier-2 as it would appear on the ODT (including removal of extraneous blanks if applicable). Each line ends with a carriage return and line feed characters; the last line also ends with an ETX character. Identifier-2 must be an alphanumeric data item of at least 80 characters. If identifier-2 is too small for the response (return message), a null character HEX(00) follows the last full line inserted. If no line can be inserted, the first character of identifier-2 is null.

CALL "INTRINSIC GETMCP" passes the processor number (the processor number can be obtained from the JOBINFO SDI) in identifier-1 to the MCP. In return, the MCP passes back a structure containing the the following:

- TIME OF RESPONSE (milliseconds)
- RLOG next record pointer
- SLOG next record pointer
- MLOG next record pointer
- RLOG next file number
- SLOG next file number

- MLOG next file number
- HOSTNAME for that particular processor

In the USING phrase of CALL "INTRINSIC GETMCP", identifier-1 must be an unsigned numeric data item of two digits specifying the processor number.

In the GIVING phrase of CALL "INTRINSIC GETMCP", identifier-2 must be an alphanumeric data item of at least 76 digits. The data returned in identifier-2 conforms to the following format:

```
02  IDENTIFIER-2.
    03  RESERVED                       PIC X(2).
    03  TIME-OF-RESPONSE               PIC 9(8)   COMP.
    03  RLOG-NEXT-RECORD-POINTER       PIC 9(8)   COMP.
    03  SLOG-NEXT-RECORD-POINTER       PIC 9(8)   COMP.
    03  MLOG-NEXT-RECORD-POINTER       PIC 9(8)   COMP.
    03  RLOG-NEXT-FILE-NUMBER          PIC 9(2)   COMP.
    03  SLOG-NEXT-FILE-NUMBER          PIC 9(2)   COMP.
    03  MLOG-NEXT-FILE-NUMBER          PIC 9(2)   COMP.
    03  HOSTNAME                       PIC X(17).
```

Format 9

CALL "INTRINSIC ZIPSPO" allows the calling program to pass control information to the MCP and to determine if any errors occurred during processing.

In the USING phrase of CALL "INTRINSIC ZIPSPO", literal-1 or identifier-1 must be an alphanumeric data item. Unless the first character of literal-1 or identifier-1 is a period, literal-1 or identifier-1 must have fewer than 72 characters. Furthermore, the input must end with a period. If a period is not present in the first 72 characters of the input, the MCP inserts a period into input location 72. If the first character of the input is a period, the string can exceed 72 bytes, and the input must end with a period.

You cannot use a ZIPSPO statement with the following keyboard input messages:

BO	BT	DD	DL	DQ
ED	ES	ET	HL	ID
KX	LH	LO	QD	XA

You cannot use a ZIPSPO statement with the following MCP control instructions:

ALLOCATE	ALTER	BCL	BINARY
DATA(B)	DEBUG	DISK	DISPLAY
DLP	EBCDIC	LABEL1	PACK
PATCH	QUIKPOOL	QWKMEM	SHOW
UNIT	WHATS		

In the GIVING phrase of CALL "INTRINSIC ZIPSPO", identifier-2 must be an alphanumeric data item of 82 characters in length. The first two characters are numeric error code (CTLERR). This 82 character variable provides error information.

If no error occurs, the first 2 characters of the error code are "00". The next 4 characters contain the run log number of the last job initiated by the ZIPSPO BCT or "0000" if no job was initiated. The last 76 characters are undefined.

If an error occurs, the first 2 characters of the error code are non-zero and the next 80 characters contain the error message.

If the input syntax is correct but the operation cannot be performed by the MCP, no response is returned.

Format 10

CALL "INTRINSIC BINARY-DECIMAL" reads the binary data specified by literal-1 or identifier-1 and converts the entire value to its decimal representation that is stored in the GIVING phrase, identifier-2.

Literal-1 must be a hexadecimal literal not to exceed 100 digits in length. Identifier-1 must be an unsigned computational numeric data item between 1 to 100 digits in length. If literal-1 or identifier-1 is greater than 100 digits, the value is truncated and the first 100 digits are used.

If the size of the converted data is less than identifier-2, the converted data is stored, right justified with leading zeros. Identifier-2 must not be longer than 100 units, and must be defined as an unsigned numeric data item.

If the size of the converted data is greater than identifier-2, and the source data is not zero; the converted data cannot be assigned to identifier-2.

CALL "INTRINSIC DECIMAL-BINARY" reads the decimal data specified by literal-1 or identifier-1, converts the entire value to binary, and stores the binary value in identifier-2.

Literal-1 must be a numeric literal whose characters are selected from digits 0 through 9. The length must not be greater than 100 units. If identifier-1 or literal-1 is an integer, the digits are treated as characters by assuming numeric zone digits.

If the size of the converted data is less than identifier-2, the converted data is stored right justified with leading zeros. Identifier-2 must not be greater than 100 digits. It must be defined as an unsigned computational numeric item.

If the size of the converted data is greater than identifier-2, and the source data is not zero, the converted data is assigned to identifier-2.

CALL "INTRINSIC TRANSLATE" translates a string of digits or characters according to a specified equivalence table and stores the translated string in a specified address.

In the USING phrase of CALL "INTRINSIC TRANSLATE", identifier-3 can be up to 10,000 digits or characters long. If identifier-3 or literal-3 is signed, the sign is ignored. Identifier-4 specifies the equivalence table. Identifier-4 must be alphanumeric or numeric display without editing PICTURE symbols. The translated data is stored in identifier-5. The data must be alphanumeric or unsigned numeric. The size must be greater than or equal to the size of identifier-3 or literal-3.

The following example shows the coding used to perform binary to decimal and decimal to binary conversions and translation between character sets:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
77  DEC-VAL            PIC 9(18) COMP.
77  I-STRING           PIC 9(5)  COMP VALUE 7.
77  O-STRING           PIC 9(5)  COMP.
01  MAP-TABLE.
    03  TRN            PIC 9(1000).
01  BIN-STRING.
    03  BIN-FIL1       PIC X(4) VALUE @ABCDEF78@.
    03  BIN-FIL2  REDEFINES BIN-FIL1.
        05  BIN-VAL    PIC 9(8)  COMP.
        :
        :
PROCEDURE DIVISION.
START-CONV.
    MOVE 0 TO DEC-VAL.
    MOVE ALL @F5@ TO MAP-TABLE.
    CALL "INTRINSIC BINARY-DECIMAL" USING BIN-VAL GIVING DEC-VAL.
    :
    :
    CALL "INTRINSIC DECIMAL-BINARY" USING "174" GIVING BIN-VAL.
    :
    :
    CALL "INTRINSIC TRANSLATE" USING I-STRING, MAP-TABLE
                                          GIVING O-STRING.
    :
    :
    CALL "INTRINSIC TRANSLATE" USING "7", MAP-TABLE
                                          GIVING O-STRING.
    :
    :
```

Format 11

CALL "INTRINSIC GETLABEL" causes the value of the label area for file-name-1 to be moved to identifier-1.

CALL "INTRINSIC PUTLABEL" causes the value in identifier-2 to be moved to the label area for file-name-2.

File-name-1 and file-name-2 must be tape files. Identifier-1 and identifier-2 must be WORKING-STORAGE data items.

These system dependent intrinsics (SDIs) provide access to the tape label area without the overhead of USE procedures. The GETLABEL SDI is most useful for accessing the label data of an input tape file after the file has been opened.

The PUTLABEL SDI does restrict what can be added or modified in tape labels. Specifically, nothing in the first 80 characters of the label area for a USASI label or selected fields as defined in Table 5–2 for a STANDARD label, can be modified. The label area must be set up prior to opening or closing the file rather than being set during the opening or closing of a file in a USE procedure.

An example of a program that accesses tape labels is included in Appendix E.

CALL MODULE Statement

The CALL MODULE statement allows you to invoke independently compiled modules (ICMs), using the COBOL user program.

The following is the general format:

```
CALL MODULE [ "section-name" OF ] "program-name"

     FROM { "ICM-file-name" } [ ON { "pack-name" } ]
          { ICMFILE         }      { DISK      }
```

CALL MODULE permits a compilation to include sections from other independently compiled modules (ICMs). (An ICM is created during a COBOL compilation when the dollar option ICM is set.) CALL MODULE can appear anywhere in the Procedure Division. The referenced module can be a program or any nondeclarative section within a program.

CALL MODULE transfers control from the statement containing CALL MODULE to the first statement in the program or section named in the CALL statement.

Section-name, program-name, ICM-file-name, and pack-name must be alphanumeric literals. That is, each must appear in quotation marks.

CALL MODULE program-name transfers control to the first nondeclarative, executable statement in program-name.

CALL MODULE section-name OF program-name transfers control to the first executable statement in the section named section-name within program-name.

Control remains with the called program until an EXIT MODULE is executed, at which point control returns to the next executable statement following CALL MODULE.

The program-name must consist of the first six characters of the PROGRAM-ID of the called program.

The ICM-file-name is the name of the independently compiled module created during the compilation of the program-name. It is the name following ICM in the dollar record in that compilation.

ICMFILE specifies that the ICM file name is specified in the ICMFILE dollar option.

The pack-name, if used, is the name of the diskpack on which the ICM file resides.

The following rules designate the location of the called ICM file:

- When you specify the ICMFAMILY dollar option and omit the ON DISK clause, the ICM file is assumed to reside on the 180-byte diskpack media specified in the ICMFAMILY dollar option.
- When you specify neither the ICMFAMILY dollar option nor the ON clause, the ICM file is assumed to reside on 100-byte disk media.
- If you specify both the ICMFAMILY dollar option and the ON clause, the ICM file is assumed to reside on the media specified in the ON clause.

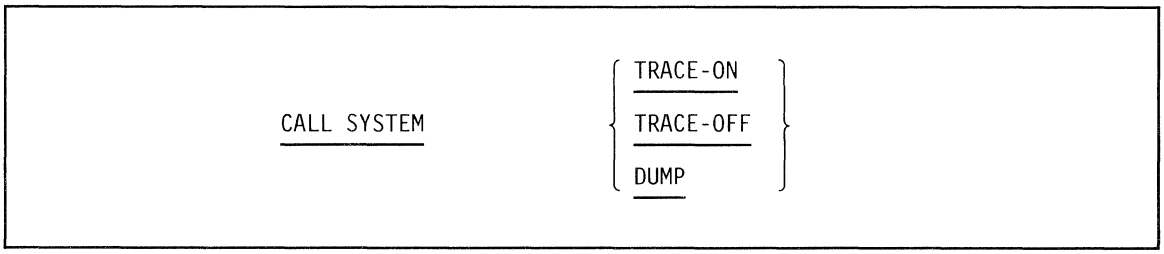
Refer to Section 10, "COBOL Binding" for further information.

CALL SYSTEM Statement

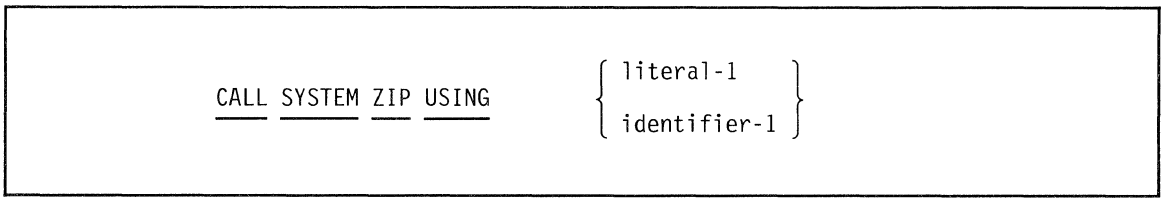
The CALL SYSTEM statement allows you to invoke system utility functions from the COBOL user program.

The following are the general formats:

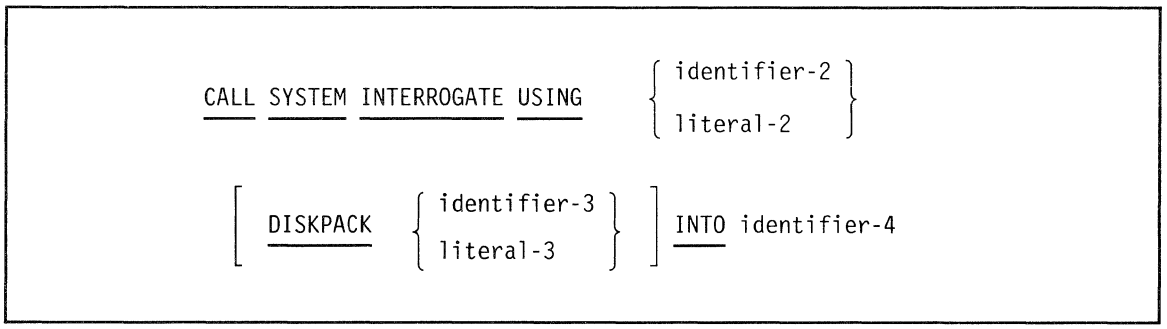
Format 1



Format 2



Format 3



Format 1

The CALL SYSTEM TRACE-ON traces all normal mode instructions until it encounters the CALL SYSTEM TRACE-OFF.

The CALL SYSTEM TRACE-OFF turns the trace off.

The CALL SYSTEM DUMP causes a program memory dump. Processing continues after the memory "snapshot."

Format 2

The CALL SYSTEM ZIP construct causes the operating system to execute an MCP control instruction contained within the object program. You can use ZIP for programmatic scheduling of programs or for accomplishing any of the MCP control functions that can be initiated through the ODT or card reader.

Literal-1 must be a nonnumeric literal, containing information that could be entered as one or more MCP control instructions. You must enclose literal-1 with quotation marks. In addition, literal-1 must end with a period. If literal-1 is longer than 72 characters, it must also begin with a period.

The following example shows the syntax for a literal shorter than 72 characters:

```
CALL SYSTEM ZIP USING "EXECUTE PROG2."
```

Note that the ZIP string ends with a period and is enclosed in quotation marks.

The following example shows the syntax for a literal longer than 72 characters:

```
CALL SYSTEM ZIP USING  ".EXECUTE PROG2; INSERT 0 11
   ""INSERT THIS""; FILE LINE = PROGPR; AFTER PROG1."
```

Note that the ZIP string both begins and ends with a period, and is enclosed in quotation marks. Quotation marks within the string are represented by double quotation marks.

Identifier-1 must refer to an alphanumeric data item that contains a value equivalent to one or more MCP control instructions.

Caution

A ZIP string not ending with a period can cause unpredictable results.

Format 3

The CALL SYSTEM INTERROGATE causes the MCP to search the directory for the file specified by identifier-2 or by literal-2.

If you select the optional DISKPACK clause, the MCP searches the directory of the diskpack whose multi-file id is specified by identifier-3 or literal-3. If the clause is absent, the MCP searches the disk directory.

The result of the search is returned as a one-digit response and placed in identifier-4. If the file is present, a one is returned; if not, a zero is returned. If identifier-4 is not described as PIC 9 COMP, the appropriate data conversion and justification is done.

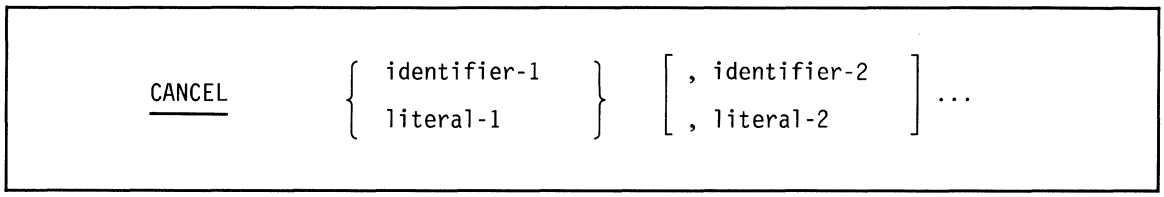
Identifier-2, identifier-3, literal-2, and literal-3 must all be 6-character alphanumeric items and must exactly match the desired file-id and multi-file id.

When searching for a file on a resource pack, you should include the DISKPACK option with identifier-3 or literal-3 containing 6 spaces.

CANCEL Statement

The CANCEL statement releases the memory areas occupied by the program called in a CALL statement (refer to "CALL Statement" in this section for details).

The following is the general format:



You must define identifier-1 and identifier-2 as alphanumeric data items with values consistent with program-naming conventions (that is, 6-character filenames) and define literal-1 and literal-2 as nonnumeric literals.

Cancel a called program by referring to it in a CANCEL statement or by terminating the run unit associated with the called program.

The program or programs specified in the CANCEL statement cannot refer to a program that has not yet executed an EXIT PROGRAM statement. Only a subsequent CALL statement can establish the logical relationship to a cancelled program.

No action is taken when a CANCEL statement specifying a non-called program or a cancelled called program is executed; however, control passes to the next statement.

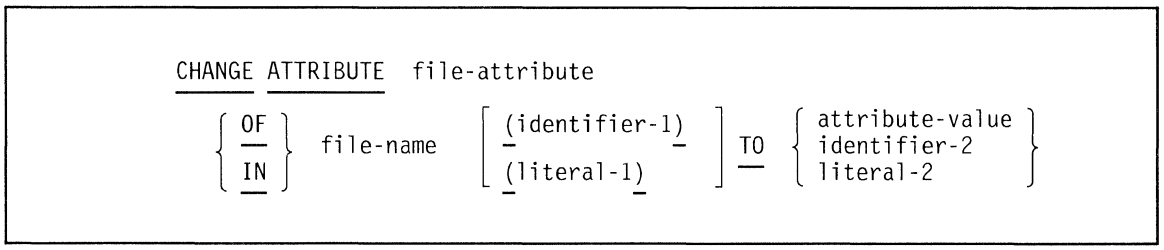
After a CANCEL statement is executed, the referenced program has no logical relationship to the program containing the CANCEL statement; however, a subsequent CALL statement specifying that program initializes the program to its previous state. Consequently, the memory areas associated with the called program are released and are available to the system.

Note: *The $LEV2IPC dollar-option must be set when you use the CANCEL statement.*

CHANGE Statement

The CHANGE statement allows you to modify or redefine a file attribute from the user program.

The following is the general format:



You can use the CHANGE ATTRIBUTE statement to modify any file-attribute that can be written (as identified in Table 7–1 in Section 7), with the following exceptions: you cannot change the file attributes ACCESSMODE, BLOCKSTRUCTURE, AREAS, and AREASIZE for relative or indexed files; any attempt to do so causes a syntax error.

Use the OF or IN clause to specify the file-name of the file of the corresponding attribute. When you use CHANGE to modify a subport file attribute, identifier-1 or literal-1 represents the subport-index of the file. The value of a subport-index is an integer variable or constant whose range is determined by the port file's MAXSUBFILES attribute. Table 7–2 summarizes a complete list of port and subport file attributes.

The following example shows a valid CHANGE statement that accesses the subport file attribute YOURNAME of the port file PORTA:

```
CHANGE ATTRIBUTE YOURNAME OF PORTA (2) TO "PRTB".
```

If the file attribute to be changed is a mnemonic file attribute, then the TO clause must specify an associated attribute-value exactly as listed in column 2 of Table 7–1. Do not use identifier-2 or literal-2.

For numeric and alphanumeric file attributes, the TO clause must specify an identifier or literal whose value corresponds to the associated range listed in column 2 of Table 7–1.

For example, the following are valid CHANGE statements for attributes of a file named TESTOR:

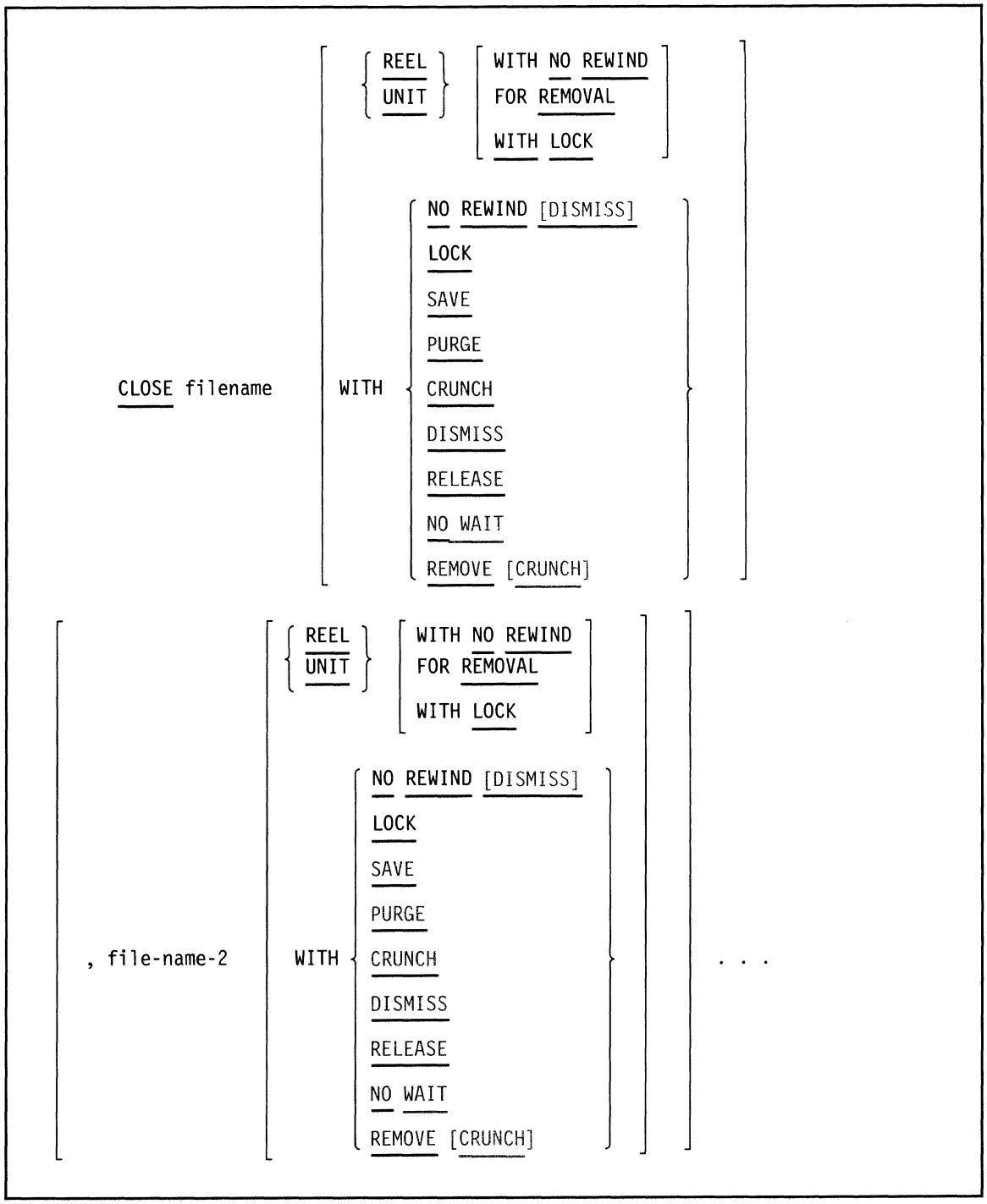
```
CHANGE ATTRIBUTE ACCESSMODE OF TESTOR TO SEQUENTIAL.
CHANGE ATTRIBUTE AREAS OF TESTOR TO 20.
CHANGE ATTRIBUTE CYLINDERMODE OF TESTOR TO TRUE.
CHANGE ATTRIBUTE FAMILYNAME OF TESTOR TO "PACK01".
CHANGE ATTRIBUTE UNITNO OF TESTOR TO "999".
```

Section 7 contains a complete discussion of the usage and restrictions of all file attributes.

Note: *When the FD of a file contains the declaration "VALUE OF FILENAME IS identifier", the contents of the identifier have priority over the contents of the TO clause in a CHANGE ATTRIBUTE FILENAME statement. That is, the program first executes the CHANGE statement, but then uses the contents of the identifier in the VALUE OF clause when the file is opened. In turn, an MCP FILE statement at program execution time has priority over any file name specified.*

CLOSE Statement

The CLOSE statement terminates the processing of a file or a reel or unit of a file. It can also specify the disposition of the file and device to which the file is assigned. The following is the general format:


CLOSE filename
[{ REEL / UNIT } [WITH NO REWIND / FOR REMOVAL / WITH LOCK]
[WITH { NO REWIND [DISMISS] / LOCK / SAVE / PURGE / CRUNCH / DISMISS / RELEASE / NO WAIT / REMOVE [CRUNCH] }]
[, file-name-2 [{ REEL / UNIT } [WITH NO REWIND / FOR REMOVAL / WITH LOCK]
[WITH { NO REWIND [DISMISS] / LOCK / SAVE / PURGE / CRUNCH / DISMISS / RELEASE / NO WAIT / REMOVE [CRUNCH] }]]] . . .

The files referenced in the CLOSE statement need not all have the same organization or access.

A CLOSE statement can only be executed for a file in an open mode and the REEL or UNIT phrase must be used only for sequential files.

Except where otherwise stated in the following general rules, the terms REEL and UNIT are synonymous and completely interchangeable in the CLOSE statement. Also, the terms DISMISS and RELEASE are synonymous. The statements CLOSE filename RELEASE and CLOSE filename LOCK must end with either a semicolon or a period.

If a file is open when a STOP RUN statement is executed or when a termination occurs, the file is closed as if a simple CLOSE statement had been executed.

If you specify the OPTIONAL phrase for the file in the FILE-CONTROL paragraph of the Environment Division and the file is not present, standard end-of-file processing is not performed for that file.

If a CLOSE statement without the REEL or UNIT phrase is executed for a file, no other statement (except the SORT or MERGE statements with the USING or GIVING phrases) can be executed that references that file, until the file is opened.

The WITH NO REWIND, FOR REMOVAL, SAVE, PURGE, CRUNCH, DISMISS, and RELEASE phrases have no effect at object time if they do not apply to the storage medium on which the file resides.

Executing a CLOSE statement has no effect upon the contents or the availability of the file's record area.

You can specify the WITH NO WAIT phrase only for port files.

A CLOSE statement without a phrase specified causes the program to wait until the file is closed before resuming execution. This wait is prevented for port files by specifying the WITH NO WAIT phrase, which returns control to the next statement without waiting for the CLOSE to be completed.

A CLOSE statement with the REEL LOCK or UNIT LOCK phrase specified allows the file to be reopened when that run unit is executed.

The results of executing each type of CLOSE for different categories of files are summarized in Table 6–4. The definitions of the numbers used in Table 6–4 follow the table.

Table 6–4. Relationship of Files and the Formats of the CLOSE Statement

Device	Mass Storage		Tape (Sequential)				Other	
			Single Reel		Multiple Reel			
CLOSE Option	**Old**	**New**	**Input**	**Output**	**Input**	**Output**	**Input**	**Output**
CLOSE	1, 5	3, 5	5, 10, 12	5, 10, 12	5, 10, 12, 15, 16	5, 10, 12, 15	6, 8	6, 8, 12
REEL/UNIT	X	X	X	X	10, 14, 17	10, 14, 17, 18	X	X
REEL/UNIT NO REWIND	X	X	X	X	11, 14, 17	11, 14, 17, 18	X	X
REEL/UNIT FOR REMOVAL	X	X	X	X	7, 10, 14, 17, 19	7, 10, 14, 17, 18, 19	X	X
NO REWIND	X	X	5, 11, 12	5, 11, 12	5, 11, 12, 15, 16	5, 10, 12, 15	X	X
NO REWIND DISMISS	X	X	6, 7, 11, 12	6, 7, 11, 12	6, 7, 11, 12, 15, 16	6, 7, 11, 12, 15	X	X
LOCK	1, 22	2, 22	7, 9, 10, 12, 22	7, 9, 10, 12, 22	7, 9, 10, 12, 15, 16, 22	7, 9, 10, 12, 15, 22	X	X
SAVE	1, 6	2, 6	X	X	X	X	X	X
REEL/UNIT LOCK	X	X	7, 10, 12, 23	7, 10, 12, 23	7, 10, 12, 17, 19, 23	7, 10, 12, 17, 19, 23	X	X
PURGE	4	4	7, 10, 12, 13	7, 10, 12, 13	7, 10, 12, 13, 15, 16	7, 10, 12, 13, 15	X	X
CRUNCH	1, 6, 20	2, 6, 20	X	X	X	X	X	X
DISMISS/ RELEASE	1, 6, 7	2, 6, 7	6, 7, 10, 12	6, 7, 10, 12	6, 7, 10, 12, 15, 16	6, 7, 10, 12, 15	6, 7	6, 7
WITH REMOVE	1, 6	2, 6, 21	X	X	X	X	X	X
WITH REMOVE CRUNCH	1, 6, 20	2, 6, 20, 21	X	X	X	X	X	X

Notes to Table 6–4:

1. File remains in directory.
2. File is entered in directory (becomes a permanent file).
3. File is not entered in directory.
4. File is removed from directory.
5. Association between logical and physical file is retained.
6. Association between logical and physical file is severed, and file is marked available.
7. Device is released to system.
8. Device is not released to system.
9. Unit is made not ready.
10. Current reel is rewound.
11. Current reel is not rewound.
12. If the file is positioned at its end, labels are processed (if specified).
13. Scratch label is written (if write ring is present).
14. Standard reel/unit swap takes place.
15. Standard reel/unit swap takes place for all reels/units in the file prior to the current reel/unit except for those controlled by a prior CLOSE REEL/UNIT statement.
16. If the current reel/unit is not the last in the file, the reels/units following the current one are not processed.
17. Beginning reel/unit label procedure takes place.
18. Ending reel/unit label procedure takes place.
19. Current reel/unit can be accessed again in its proper order of reels/units within the file if a subsequent CLOSE without the REEL/UNIT phrase is executed followed an OPEN statement.
20. Any unused portions of mass storage areas allocated for the file are released to the system.
21. If another file of the same name exists in the directory, the old file is removed without operator intervention.
22. File is marked so it cannot be reopened by the same program. Any attempt to do so results in a DS or DP condition and the MCP message "INVALID OPEN - FILE NOT CLOSED ".
23. File is marked so it can be reopened by the same program.

COMPUTE Statement

The COMPUTE statement assigns the value of an arithmetic expression to one or more data items.

The following is the general format:

```
COMPUTE identifier-1 [ROUNDED] [,identifier-2 [ROUNDED] ] ...

    = arithmetic-expression [; ON SIZE ERROR imperative-statement]
```

An arithmetic expression consisting of a single identifier or literal provides a method of setting the values of identifier-1 and identifier-2 so they are equal to the value of the single identifier or literal (see "Arithmetic Expressions" in this section).

If more than one identifier is specified for the result of the operation (that is preceding =), the value of the arithmetic expression is computed and then stored as the new values of identifier-1 and identifier-2 respectively.

Identifiers that appear only to the left of = must refer to either an elementary numeric item or an elementary numeric-edited item.

Additional rules and explanations relative to this statement appear in "Intermediate Data Item", "ROUNDED Phrase", "SIZE ERROR Phrase", "Overlapping Operands", and "Multiple Results in Arithmetic Statements" in this section.

Caution

The size of an intermediate data item in an arithmetic operation is limited to 98 digits (see "Intermediate Data Item" in this section). When coding COMPUTE statements, take care to design the statement so that it requires no intermediate fields longer than 98 digits.

A multiply requires a field length equal to the sum of the lengths of the operands; an addition requires a length one digit larger than the largest operand; exponentiation requires a field length equal to the length of the base times the power; subtraction and division require a result field equal in length to the largest operand.

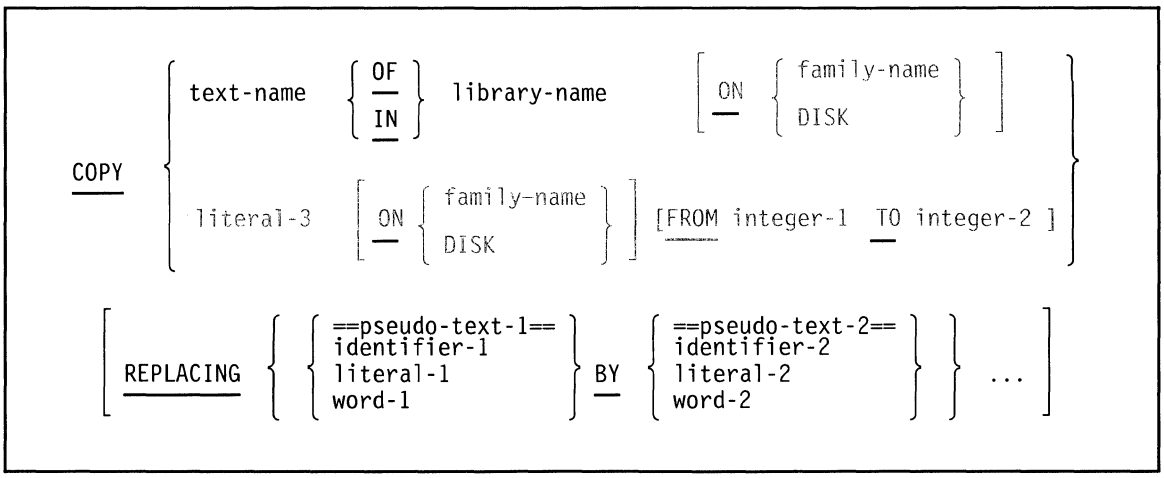
For example, a statement of the type *A = A * B + C* requires an intermediate field whose length is equal to the combined lengths of *A* and *B*. If that length is greater than 98 digits, unexpected results can occur. When such an expression must be evaluated, and the combined length of *A* and *B* is greater than 98 digits, the expression should either be coded in two COMPUTE statements (*COMPUTE A = A * B*, *COMPUTE A = A + C*), or coded in separate MULTIPLY and ADD statements (*MULTIPLY B BY A*, *ADD C TO A*).

By using separate statements, you can add ON SIZE ERROR clauses to check for an overflow of significant data. An ON SIZE ERROR clause checks only for an overflow of significant data as it is moved into the output field; it does not check for an 98-digit limit overflow in any intermediate data items.

COPY Statement

The COPY statement incorporates text into a COBOL source program.

The following is the general format:



Note: *Although a COPY statement can appear in any division of a COBOL program, COPY is considered a COBOL verb in the Procedure Division.*

A COPY statement can occur in the source program anywhere a character-string or a separator can occur except within a COPY statement. A space must precede the COPY statement, which must end with a separator period.

Compiling a source program containing COPY statements is logically equivalent to processing all COPY statements prior to processing the resulting source program.

Processing a COPY statement causes the library text associated with text-name to be copied into the source program, logically replacing the entire COPY statement, beginning with the reserved word COPY and ending with the period.

Library text must conform to the rules for COBOL reference format. The syntactic correctness of the library text cannot be independently determined. The syntactic correctness of the entire COBOL source program cannot be determined until all COPY statements have been completely processed.

Additional information on library text directly follows the discussion of this statement.

Text-name is a user-defined word (that is, the length must be equal to or less than 30 characters).

Library-name and literal-3 identify a file that is to be copied. Literal-3 must be a nonnumeric literal. It is logically the same as library-name except that it appears in quotation marks and must be a valid COBOL literal. The following rules apply to the ON clause:

- The ON family-name clause specifies the name of the 180-byte diskpack that has the library file to be copied.
- The ON DISK clause specifies that the library file resides on 100-byte disk.
- If you do not use the ON clause, the library file is assumed to reside on the diskpack specified in the LIBRFAMILY dollar option.
- If you do not use the ON clause, and the LIBRFAMILY dollar option is not present, the library file is assumed to reside on disk.

Debugging lines are permitted within library text. If a COPY statement is specified on a debugging line, then the resulting text appears as though it has been specified on debugging lines, with the following exception: comment lines in library text appear as comment lines in the resultant source program.

Text resulting from a COPY statement must not contain a COPY statement.

The optional FROM clause provides a method of copying only a portion of a source file. Integer-1 can be zero. Copying begins in the library file at the first sequence number that is equal to or greater than integer-1; copy terminates at the first sequence number that is greater than integer-2. If you do not specify integer-2, the COPY continues until the compiler reaches the end of the library file. Using the FROM phrase is not permitted with the qualified text-name.

REPLACING Clause

If you do not include the REPLACING phrase, the library text is copied unchanged. If you include the REPLACING phrase, the library text is copied; each properly matched occurrence of pseudo-text-1, identifier-1, word-1, and literal-1 in the library text is replaced by the corresponding pseudo-text-2, identifier-2, word-2, or literal-2. However, the source listing does not reflect the results of the REPLACING. Rather, the original library file appears in the source listing.

A comment line occurring in the library text and pseudo-text-1 is interpreted, for purposes of matching, as a single space. Comment lines appearing in pseudo-text-2 and library text are copied unchanged into the source program.

Pseudo-text-1 must not be null, consist solely of the character space(s), nor consist solely of comment lines. Pseudo-text-2 can be null.

Character-strings within pseudo-text-1 and pseudo-text-2 can be continued. However, both characters of a pseudo text delimiter must be on the same line (see "Coding Form" in Section 1).

Word-1 or word-2 can be any single COBOL word.

For purposes of matching, identifier-1, word-1, and literal-1 are treated as pseudo text containing only identifier-1, word-1, or literal-1 respectively.

The following are examples of pseudo text, identifier-1, word-1, and literal-1:

Pseudo text

```
==MOVE ALL "ABC"==
==EMPLOYEE-NO(INDX)==
```

Identifier-1

```
EMPLOYEE-NAME
EMPLOYEE-NO(INDX)
```

Word-1

```
MOVE
ALL
```

Literal-1

```
"MOVING ON"
123
```

The following are examples of valid REPLACING clauses:

```
REPLACING ==MOVE ALL "ABC"==BY ==ADD 3==
REPLACING EMPLOYEE-NO(INDX) BY "REPLACED"
REPLACING MOVE BY "MOVE ALL"
REPLACING "MOVING ON" BY ====
```

The comparison operation to determine text replacement occurs as follows:

1. Any separator comma, semicolon, and/or space(s) preceding the leftmost library text-word is copied into the source program. Starting with the leftmost library text-word and the first pseudo-text-1, identifier-1, word-1, or literal-1 that was specified in the REPLACING phrase, the entire REPLACING phrase operand that precedes the reserved word BY is compared to an equivalent number of contiguous library text-words.
2. Pseudo-text-1, identifier-1, word-1, or literal-1 match the library text if the ordered sequence of text-words from pseudo-text-1, identifier-1, word-1, or literal-1 is equal, character for character, to the ordered sequence of library text-words. For purposes of matching, each occurrence of a separator comma or semicolon in pseudo-text-1 or in the library text is considered to be a single space except when pseudo-text-1 consists solely of either a separator comma or semicolon. In this case, a single comma or semicolon participates in the match as a text-word. Each sequence of one or more space separators is considered to be a single space.
3. If no match occurs, the comparison is repeated with each next successive pseudo-text-1, identifier-1, word-1, or literal-1, if any, in the REPLACING phrase until either it finds a match or there is no next successive REPLACING operand.

4. When all the REPLACING phrase operands have been compared and no match has occurred, the leftmost library text-word is copied into the source program. The next successive library text-word is then considered as the leftmost library text-word. The comparison cycle starts again with the first pseudo-text-1, identifier-1, word-1, or literal-1 specified in REPLACING phrase.
5. Whenever a match occurs between pseudo-text-1, identifier-1, word-1, or literal-1 and the library text, the corresponding pseudo-text-2, identifier-2, word-2, or literal-2 is placed into the source program. The library text-word immediately following the rightmost text-word that participated in the match is then considered as the leftmost library text-word. The comparison cycle starts again with the first pseudo-text-1, identifier-1, word-1, or literal-1 specified in the REPLACING phrase.
6. The comparison operation continues until the rightmost text-word in the library text has either participated in a match or been considered as a leftmost library text-word and thus has participated in a complete comparison cycle.

For purposes of compilation, text-words after replacement are placed in the source program according to the rules for reference format (see "Coding Form" in Section 1).

When replacing entries with a COPY statement, note the following:

- If replacement of a PICTURE string is required, then the word PICTURE (or PIC) must precede the string to be replaced. For example, the following statement would be correct to replace the X(3) in PIC X(3) by X(4):

```
COPY library-name  REPLACING
    == PIC   X(3) ==  BY
    == PIC   X(4) ==.
```

- Commas and semicolons are treated exactly as spaces. A comma or semicolon by itself, therefore, cannot be replaced. Doing so would be meaningless since the listing does not include replaced text and commas and semicolons are semantically equivalent to spaces. If an attempt is made to replace just a comma or semicolon, an error occurs, because the semicolon in pseudo-text-1 is treated as a space, and pseudo-text-1 cannot consist entirely of spaces.
- If no occurrence of a particular pseudo-text-1, identifier-1, literal-1, or word-1 is in the library file, a warning is issued identifying the item that was not replaced. This warning is issued to aid in determining whether the desired replacements actually took place. If this warning is issued, check for misspellings, and verify that the REPLACING clause is correct.
- No part of pseudo-text-1, pseudo-text-2, literal-1, and so forth, should be in area A. If it is, results become unpredictable. If you want to replace a word in area A, put that word first in pseudo-text-1 or use it by itself. For example, either of the following program segments will have the expected results:

```
PARA-1.
    MOVE A TO B.
       ... REPLACING PARA-1 BY PARA-2 ...

PARA-1.
    MOVE A TO B.

       ... REPLACING  == PARA-1.  MOVE ==
           BY  == PARA-2.  MOVE == ...
```

Library

The library feature provides a means of copying text from a "library" of source text material that is available at compile time. A short phrase can put large amounts of source library material into the source program, thus saving repetitious coding. Once established, a source library can often be referenced by many programs.

Creating Libraries

COBOL library files can be created and maintained in several ways. COBOL source text can be stored on disk or diskpack as source library files. A library file can be created by any program (for example, the system utility DMPALL) capable of creating a disk or diskpack file with a record length of 80 bytes and any blocking factor between 1 and 9 inclusive. The file consists of source images only.

Texts within library programs must be delimited by $COPYBEGIN and $COPYEND, each followed by text-name.

The following card input DMPALL specification creates a library file named "MASTER" on disk:

```
?PFM CRDDSK MASTCD 80 1 MASTER 80 9 CRUNCH
?DATA MASTCD
... (COBOL source records) ...
?END
```

Note: *Use the CRUNCH option to prevent wasting space on disk or diskpack. Refer to the* V Series Systems Operations Guide, Volume 3: System Utilities *for information on DMPALL.*

You can create library files directly from CANDE system EDITOR files with the COPY command. The COBOL statements for one or more library entries are maintained in an EDITOR file. You can use the following commands to create library files from the EDITOR file. The first command creates a library file on disk; the second creates a library file on diskpack.

```
COPY sequence-range TO DISK AS library-name  9

COPY sequence-range OF editor-file-name
    TO PACK family-name AS library-name
```

In the first case, the currently edited file is copied. For the copy to pack, the file must be released. Refer to the *V Series CANDE Installation and Operations Guide* for further information.

All library files residing on disk must have unique text-names. All library files residing on the same diskpack family-name or on unrestricted (unnamed) diskpacks must have unique names.

A single library file can optionally contain multiple texts (named sections of a library). These library texts are delimited within the library as follows:

```
$COPYBEGIN      text-name
          .
          .
          .
          .
$COPYEND        text-name
```

If no text-name appears in the COPY statement, COPYBEGINs and COPYENDs are ignored. Similarly, $COPYBEGIN and $COPYEND in the primary source are also ignored.

Using Libraries

COBOL libraries contain library texts that are available to the compiler for copying at compile time. A COPY statement causes text to be inserted into the source program, where the compiler treats it as part of the source program. Copied text does not become a permanent part of the source program if a new source file is created; only the COPY statement is entered into the new source file.

The COPY statement can either copy from a library file without change, or replace all occurrences of a given literal, identifier, or word in the library text, without altering text, during the copying process.

More than one COBOL library file can be used in a compiled program. Copied text cannot contain a COPY statement.

DELETE Statement

The DELETE statement logically removes a record from a relative or indexed file.

The following is the general format:

```
DELETE file-name RECORD
    [  ; INVALID KEY imperative-statement ]
```

Do not use the INVALID KEY phrase in a DELETE statement that references a file in sequential access mode. Use the INVALID KEY phrase for a DELETE statement that references a file that is not in sequential access mode and does not include an applicable USE procedure.

The following are the general rules:

The associated file must be in the I/O mode when the statement is executed.

For files in the sequential access mode, the last input/output statement executed for a file-name prior to executing the DELETE statement must have been a successfully executed READ statement. The record that was accessed by that READ statement is logically removed.

For a file in random or dynamic access mode, the record logically removed from the file is the following:

- For a relative file, the record identified by the contents of the RELATIVE KEY data item associated with file-name
- For an indexed file, the record identified by the contents of the prime RECORD KEY data item associated with file-name

If the file does not contain the record specified by the key, an INVALID KEY condition exists.

After a DELETE statement has been executed, the identified record can no longer be accessed. Execution does not affect the contents of the record area associated with file-name. Neither is the current record pointer affected by executing a DELETE statement. The execution does update the value of any FILE STATUS data item associated with file-name.

DISPLAY Statement

The DISPLAY statement transfers low volume data to an appropriate hardware device.

The following is the general format:

```
DISPLAY { identifier-1 } [ ,identifier-2 ] ...
        { literal-1    }   [ ,literal-2   ]

        [ UPON   mnemonic-name ]
```

The DISPLAY statement transfers the contents of each operand to the hardware device in the order in which they are listed.

When a DISPLAY statement contains more than one operand, the size of the sending item is the sum of the sizes associated with the operands. The values of the operands are transferred in the sequence in which the operands are encountered.

The maximum number of characters that can be transmitted is unlimited.

If the data transferred does not fit on one line, a DISPLAY LINES BCT is generated. This BCT causes multiple lines to be displayed, and the text is not prefaced by the program identification; that is, the program-id and mix number do not appear.

If the literal is numeric, then it must be an unsigned integer. Each literal can be any figurative constant except ALL. If a figurative constant is specified as one of the operands, only a single occurrence of the figurative constant is displayed.

If the DISPLAY statement does not include the UPON phrase, the operator display terminal (ODT) is used. If the DISPLAY statement does include the UPON phrase, the mnemonic-name is associated with a hardware device in the SPECIAL-NAMES paragraph in the Environment Division and must be associated with the hardware-name ODT. DISPLAY sends messages only to an ODT device. The message will appear on the ODT device of the processor on which the program is executed.

The maximum number of characters displayed on a single-line display is 60 when a program is executed on disk and 51 when the program is executed on diskpack. The reason for this limit is that the MCP precedes single-line display messages with the program's name and mix number; when executed on diskpack, the words *ON pack-name* are also present, reducing the number of character positions available on the line. The compiler assumes that 60 characters are available. If the program is to be executed on diskpack, be sure that messages are no longer than 51 characters per line.

Using a multi-level program identifier further limits the maximum number of displayable characters to 44 when using a six-character pack name and program name. For example, if a program is compiled with the statement "COMPILE JAKE01 WITH DISPLY ON LONIPK" and JAKE01/DISPLY performs a display BCT, the maximum number of characters that can be displayed on one line is 44.

DIVIDE Statement

The DIVIDE statement divides one numeric data item into others and sets the values of data items equal to the quotient and remainder.

The following are the general formats:

Format 1

```
DIVIDE { identifier-1 } INTO identifier-2 [ROUNDED]
       { literal-1    }

[, identifier-3 [ROUNDED]] ... [; ON SIZE ERROR imperative-statement]
```

Format 2

```
DIVIDE { identifier-1 } INTO { identifier-2 } GIVING identifier-3 [ROUNDED]
       { literal-1    }     { literal-2    }

 [, identifier-4 [ROUNDED]] ... [; ON SIZE ERROR imperative-statement]
```

Format 3

```
DIVIDE { identifier-1 } BY { identifier-2 } GIVING indentifier-3 [ROUNDED]
       { literal-1    }   { literal-2    }

 [,identifier-4 [ROUNDED]] ... [; ON SIZE ERROR imperative-statement]
```

Format 4

```
DIVIDE { identifier-1 } INTO { identifier-2 } GIVING identifier-3 [ROUNDED]
       { literal-1    }      { literal-2    }

    REMAINDER identifier-4 [; ON SIZE ERROR imperative-statement]
```

Format 5

```
DIVIDE { identifier-1 } BY { identifier-2 } GIVING identifier-3 [ROUNDED]
       { literal-1    }    { literal-2    }

    REMAINDER identifier-4 [; ON SIZE ERROR imperative-statement]
```

Each identifier must refer to an elementary numeric item. However, any identifier associated with the GIVING or REMAINDER phrase must refer to either an elementary numeric item or an elementary numeric-edited item. Each literal must be a numeric literal.

When Format 1 is used, the value of identifier-1 or literal-1 is divided INTO the value of identifier-2. The value of the dividend (identifier-2) is replaced by this quotient; the process is similar for identifier-1 or literal-1 and identifier-3.

When Format 2 is used, the value of identifier-1 or literal-1 is divided INTO identifier-2 or literal-2, and the result is stored in identifier-3 and identifier-4, the objects of the GIVING clause.

When Format 3 is used, the value of identifier-1 or literal-1 is divided BY the value of identifier-2 or literal-2, and the result is stored in identifier-3 and identifier-4.

Use formats 4 and 5 when you want a remainder from the division operation, namely identifier-4. The REMAINDER is the result of subtracting the product of the quotient (identifier-3) and the divisor from the dividend. If identifier-3 is defined as a numeric-edited item, the quotient used to calculate the remainder is an intermediate field that contains the unedited quotient. If ROUNDED is used, the quotient used to calculate the remainder is an intermediate field that contains the quotient of the DIVIDE statement, truncated rather than rounded.

In Formats 4 and 5, this calculation defines the accuracy of the REMAINDER data item (identifier-4). Appropriate decimal alignment and truncation (not rounding) are performed for the content of the data item referenced by identifier-4, as needed.

When the ON SIZE ERROR phrase appears in Formats 4 and 5, the following rules are relevant:

- If the size error occurs on the quotient, no remainder calculation is meaningful. Thus, the contents of the data items referenced by both identifier-3 and identifier-4 remain unchanged.
- If the size error occurs on the remainder, the contents of the data item referenced by identifier-4 remains unchanged. However, you must do your own analysis to identify the situation that has actually occurred.

Additional rules and explanations appear in the appropriate paragraphs. See "Intermediate Data Item", "Arithmetic Statements", "Overlapping Operands", "ROUNDED Phrase", "SIZE ERROR Phrase", and "Multiple Results in Arithmetic Statements" in this section.

EXIT Statement

The EXIT statement documents the logical end point for a series of sections or paragraphs that can be executed under the control of a PERFORM or CALL MODULE statement.

The following are the general formats:

Format 1

```
EXIT.
```

Format 2

```
EXIT MODULE.
```

Format 3

```
EXIT PROGRAM.
```

Format 1

An EXIT statement enables the user to assign a procedure-name to a given point in a program. An EXIT statement has no other effect on compilation or execution.

The EXIT statement must appear in a sentence by itself and must be the only sentence in the paragraph.

Format 2

An EXIT MODULE statement returns control from a called program to the calling program. The EXIT MODULE statement must appear in a sentence by itself and must be the only sentence in the paragraph. Refer to "CALL MODULE Statement" in this section and to Section 10 "COBOL Binding" for further information.

Format 3

The EXIT PROGRAM statement must appear alone in a statement, and must be the only sentence in the paragraph.

Executing an EXIT PROGRAM statement in a called program passes control to the calling program. The EXIT PROGRAM can be used only in a program that is called by another program. A program containing an EXIT PROGRAM statement cannot be executed independently unless it is a level II interprogram communication program (ICM).

The EXIT PROGRAM statement creates an ICM using the first six characters of the PROGRAM-ID name. If there is a $SET ICM record, the EXIT PROGRAM overrides the specified name and uses the program name. If the $SET ICM record also specified the packname, it is maintained.

GO Statement

The GO statement transfers control from one part of the Procedure Division to another.

The following are the general formats:

Format 1

```
GO TO [procedure-name-1]
```

Format 2

```
GO TO procedure-name-1 [, procedure-name-2] ..., procedure-name-n
      DEPENDING ON  identifier
```

When you execute a GO TO statement, represented by Format 1, control is transferred to procedure-name-1 or to another procedure-name if an ALTER statement has modified the GO TO statement.

If Format 1 does not specify procedure-name-1, an ALTER statement referring to this GO TO statement must be executed before this GO TO statement is executed. Otherwise, the program abnormally terminates. For rules concerning segment restrictions for these modified GO TO statements, refer to the "ALTER Statement," earlier in this section.

When a paragraph is referenced by an ALTER statement, that paragraph can consist of only a paragraph header and then a Format 1 GO TO statement.

A Format 1 GO TO statement without procedure-name-1 can only appear in a single statement paragraph. If a GO TO statement represented by Format 1 appears in a consecutive sequence of imperative statements, it must be the last statement in that sequence.

When you execute a GO TO statement represented by Format 2, control is transferred to procedure-name-1, procedure-name-2, ... procedure-name-n, depending on the value of the identifier being 1, 2, ..., n. If the value of the identifier is not a positive or unsigned integer from 1 through n, then no transfer occurs and control passes to the next statement in the normal execution sequence.

Identifier is the name of a numeric elementary item described without any positions to the right of the assumed decimal point.

IF Statement

The IF statement evaluates a condition to determine the subsequent action of the object program, which depends on whether the value of the condition is true or false.

The following is the general format:

```
IF condition THEN { statement-1      } { ; ELSE statement-2      }
                  { NEXT SENTENCE    } { ; ELSE NEXT SENTENCE   }
```

Condition is discussed under "Conditional Expressions" at the beginning of this section.

When you execute an IF statement the following transfers of control occur:

- If the condition is true, statement-1 is executed if specified. If statement-1 contains a procedure branching or conditional statement, control is explicitly transferred in accordance with the rules of that statement. If statement-1 does not contain a procedure branching or conditional statement, the ELSE phrase is ignored and control passes to the next executable sentence.
- If the condition is true and the NEXT SENTENCE phrase is specified instead of statement-1, the ELSE phrase is ignored and control passes to the next executable sentence.
- If the condition is false, the specified statement-1 or NEXT SENTENCE is ignored, and statement-2, if specified, is executed. If statement-2 contains a procedure branching or conditional statement, control is explicitly transferred in accordance with the rules of that statement. If statement-2 does not contain a procedure branching or conditional statement, control passes to the next executable sentence. If the ELSE statement-2 phrase is not specified, statement-1 is ignored and control passes to the next executable sentence.
- If the condition is false, and the ELSE NEXT SENTENCE phrase is specified, statement-1 is ignored and control passes to the next executable sentence.

Statement-1 and statement-2 represent either an imperative statement or a conditional statement, and either can be followed by a conditional statement. Statement-1 and/or statement-2 can contain an IF statement. In this case the IF statement is nested.

IF statements within IF statements can be considered paired IF and ELSE combinations, proceeding from left to right. Thus, any ELSE encountered is considered to apply to the immediately preceding IF that has not been already paired with an ELSE. The ELSE NEXT SENTENCE phrase can be omitted if it immediately precedes the last period in the sentence.

INSPECT Statement

The INSPECT statement allows you to tally (Format 1), replace (Format 2), or tally and replace (Format 3) occurrences of single characters or groups of characters in a data item.

The following are the general formats:

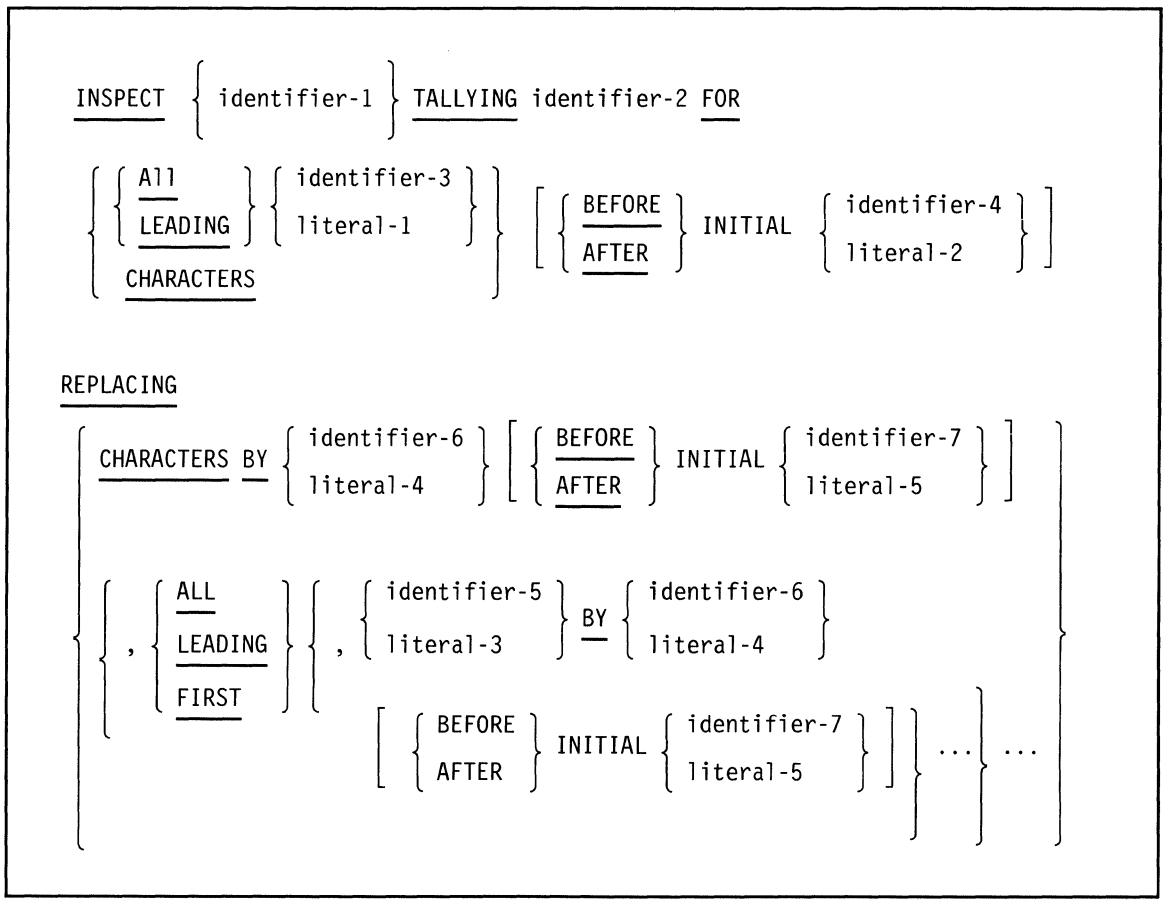
Format 1

```
INSPECT identifier-1 TALLYING identifier-2 FOR

{ { { ALL       } { identifier-3 } }                                                    }
{ { { LEADING   } { literal-1   } } [ { BEFORE } INITIAL { identifier-4 } ]              }
{ { CHARACTERS                    }   { AFTER  }         { literal-2   }               }
```

Format 2

```
INSPECT identifier-1 REPLACING

{ CHARACTERS BY { identifier-6 } [ { BEFORE } INITIAL { identifier-7 } ]                                                  }
{              { literal-4   }   { AFTER  }         { literal-5   }                                                      }
{                                                                                                                        }
{ { , { ALL       } } { , { identifier-5 } BY { identifier-6 } }                                                         }
{ { , { LEADING   } }   { { literal-3   }    { literal-4   } }                                                           }
{ { , { FIRST     } }                                                                                                    }
{                                                                                                                        }
{                                  [ { BEFORE } INITIAL { identifier-7 } ] } ... } ...                                   }
{                                  { { AFTER  }         { literal-5   } ] }     }                                        }
```

Format 3



Inspection (which includes the comparison cycle, the establishment of boundaries for the BEFORE and AFTER phrase, and the mechanism for tallying and/or replacing) begins at the leftmost character position of the data item referenced by identifier-1, regardless of its class, and proceeds to the rightmost position.

For use in the INSPECT statement, the contents of the data item referenced by identifier-1, identifier-3, identifier-4, identifier-5, identifier-6, or identifier-7 are treated as follows:

- If the identifier is described as alphanumeric, the INSPECT statement treats the contents of each such identifier as a character-string.
- If the identifier is described as alphanumeric edited, numeric edited, or unsigned numeric, the data item is inspected as though it had been redefined as alphanumeric and the INSPECT statement had been written to reference the redefined data item.
- If the identifier is described as signed numeric, the data item is inspected as though it had been moved to an unsigned numeric data item of the same length. The rules in the preceding paragraph apply.

Note: *In the following paragraphs, all references to literal-1, literal-2, literal-3, literal-4, and literal-5 apply equally to the contents of the data item referenced by identifier-3, identifier-4, identifier-5, identifier-6, and identifier-7, respectively.*

Where the contents of the data item referenced by identifier-1 are inspected, each properly matched occurrence of literal-1 is tallied (Formats 1 and 3) and/or each properly matched occurrence of literal-3 is replaced by literal-4 (Formats 2 and 3).

A Format 3 INSPECT statement is interpreted and executed as though two successive INSPECT statements specifying the same identifier-1 had been written with the following:

- One statement is a Format 1 statement with TALLYING phrases identical to those specified in the Format 3 statement.
- The other statement is a Format 2 statement with REPLACING phrases identical to those specified in the Format 3 statement.

Identifier-1 must reference either a group item or any category of elementary items, described (either implicitly or explicitly) as USAGE IS DISPLAY.

The TALLYING option causes a count to be accumulated in the data item named by identifier-2, which must be an elementary numeric item. Executing the INSPECT statement does not initalize this data item.

Identifier-3 through identifier-n must reference either an elementary alphabetic, alphanumeric, or numeric item described (either implicitly or explicitly) as usage is DISPLAY. Each literal must be nonnumeric and can be any figurative constant except ALL. If either literal-1 or literal-2 is a figurative constant, the figurative constant refers to an implicit 1-character data item. When you use a figurative constant as literal-3, the data referenced by literal-4 or identifier-6 must be one character in length.

The size of the data referenced by literal-4 or identifier-6 must be equal to the size of the data referenced by literal-3 or identifier-5. When you use a figurative constant as literal-4, the size of the figurative constant is equal to the size of literal-3 or the size of the data item referenced by identifier-5.

When you use the CHARACTERS phrase, literal-4, literal-5, or the size of the data item referenced by identifier-6, identifier-7 must be one character in length.

The following steps explain the comparison operation which determines whether the occurrences of literal-1 are to be tallied, whether occurrences of literal-3 are to be replaced, or both:

1. The operands of the TALLYING and REPLACING phrases are considered in the order they are specified in the INSPECT statement from left to right. The first literal-1, literal-3 is compared to an equal number of contiguous characters, starting with the leftmost character position in the data item referenced by identifier-1. Literal-1, literal-3 and that portion of the contents of the data item referenced by identifier-1 match if they are equal, character for character.
2. If no match occurs in the comparison of the first literal-1, literal-3, the comparison is repeated with each successive literal-1, literal-3, if any, until there is either a match or no next successive literal-1, literal-3. In the latter case, the character position in the data item referenced by identifier-1 immediately to the right of the leftmost character position considered in the last comparison cycle is considered to be the leftmost character position, and the comparison cycle begins again with the first literal-1, literal-3.
3. Whenever a match occurs, tallying and/or replacing occurs as described in the rules for tallying and replacing. The character position in the data item referenced by identifier-1, immediately to the right of the rightmost character position that participated in the match, is now considered to be the leftmost character position of the data item referenced by identifier-1. The comparison cycle then starts again with the first literal-1, literal-3.
4. The comparison operation ends when the rightmost character position of the data item referenced by identifier-1 was matched or was considered as the leftmost character position.
5. If you specify the CHARACTERS phrase, an implied 1-character operand participates in the cycle described in the previous paragraphs, except the contents of the data item referenced by identifier-1 are not compared. This implied character is considered always to match the leftmost character of the contents of the data item referenced by identifier-1.

This comparison operation is affected by the BEFORE and AFTER phrases as follows:

- If you do not include the BEFORE and AFTER phrase, literal-1, literal-3 or the implied operand of the CHARACTERS phrase participates in the comparison.
- If you include the BEFORE phrase, the associated literal-1, literal-3 or the implied operand of the CHARACTERS phrase participates only in those comparison cycles that involve that portion of the contents of the data item referenced by identifier-1 from its leftmost character position up to, but not including, the first occurrence of literal-2, literal-5 within the contents of the data item referenced by identifier-1. The position of this first occurrence is determined before the comparison operation begins. If, on any comparison cycle, literal-1, literal-3 or the implied operand of the CHARACTERS phrase is not eligible to participate, it is considered not to match the contents of the data item referenced by identifier-1. If there is no occurrence of literal-2, literal-5 within the contents of the data item referenced by identifier-1, its associated literal-1, literal-3 or the implied operand of the CHARACTERS phrase participates in the comparison operation as though the BEFORE phrase had not been specified.

- If you include the AFTER phrase, the associated literal-1, literal-3 or the implied operand of the CHARACTERS phrase can participate only in those comparison cycles that involve the portion of the contents of the data item referenced by identifier-1 from the character position immediately to the right of the rightmost character position of the first occurrence of literal-2, literal-5 within the contents of the data item referenced by identifier-1 and the rightmost character position of the data item referenced by identifier-1. The position of this first occurrence is determined before the first cycle of the comparison operation begins.

  If, on any comparison cycle, literal-1, literal-3 or the implied operand of the CHARACTERS phrase is not eligible to participate, it is considered not to match the contents of the data item referenced by identifier-1. If there is no occurrence of literal-2, literal-5 within the contents of the data item referenced by identifier-1, its associated literal-1, literal-3, or the implied operand of the CHARACTERS phrase is never eligible to participate in the comparison operation.

Tallying

The following are the rules for tallying:

- The count is accumulated in the data item named by identifier-2, which must be an elementary numeric item. The INSPECT statement does initialize identifier-2.
- If you include the ALL phrase, the contents of the data item referenced by identifier-2 increments by one for each occurrence of literal-1 matched within the contents of the data item referenced by identifier-1.
- If you include the LEADING phrase, the contents of the data item referenced by identifier-2 increments by one for each contiguous occurrence of literal-1 matched within the contents of the data item referenced by identifier-1. However, the leftmost such occurrence must be at the point from which comparison began in the first comparison cycle of literal-1.
- If you include the CHARACTERS phrase, the contents of the data item referenced by identifier-2 increments by one for each character matched, within the contents of the data item referenced by identifier-1.

Replacing

The following are the rules for replacement:

- When you include the CHARACTERS phrase, each character matched in the contents of the data item referenced by identifier-1 is replaced by literal-4.
- When you include the adjective ALL, each occurrence of literal-3 matched in the contents of the data item referenced by identifier-1 is replaced by literal-4.
- When you include the adjective LEADING, each contiguous occurrence of literal-3 matched in the contents of the data item referenced by identifier-1 is

replaced by literal-4, provided that the leftmost occurrence is where the first comparison cycle began in which literal-3 participated.

- When you include the adjective FIRST, the leftmost occurrence of literal-3 matched within the contents of the data item referenced by identifier-1 is replaced by literal-4.

Examples:

The following are six examples of the INSPECT statement:

```
INSPECT word TALLYING count FOR LEADING "L" BEFORE INITIAL "A".
```

Where word is LARGE, count is 1.
Where word is ANALYST, count is 0.

```
INSPECT word TALLYING count FOR ALL "L", REPLACING LEADING "A" BY "E"
AFTER INITIAL "L".
```

Where word is CALLAR, count is 2, word is CALLAR.
Where word is SALAMI, count is 1 and word is SALEMI.
Where word is LATTER, count is 1 and word is LETTER.

```
INSPECT word REPLACING ALL "A" BY "G" BEFORE INITIAL "X".
```

Where word is ARXAX, word is GRXAX.
Where word is HANDAX, word is HGNDGX.

```
INSPECT word TALLYING count FOR CHARACTERS AFTER INITIAL "J"
REPLACING ALL "A" BY "B".
```

Where word is ADJECTIVE, count is 6 and word is BDJECTIVE.
Where word is JACK, count is 3 and word is JBCK.
Where word is JUJMAB, count is 5 and word is JUJMBB.

```
INSPECT word REPLACING ALL "X" BY "Y", "B" BY "Z", "W" BY "Q"
AFTER INITIAL "R".
```

Where word is RXXBQWY, word is RYYZQQY.
Where word is YZACDWBR, word is YZACDWZR.
Where word is RAWRXEB, word is RAQRYEZ.

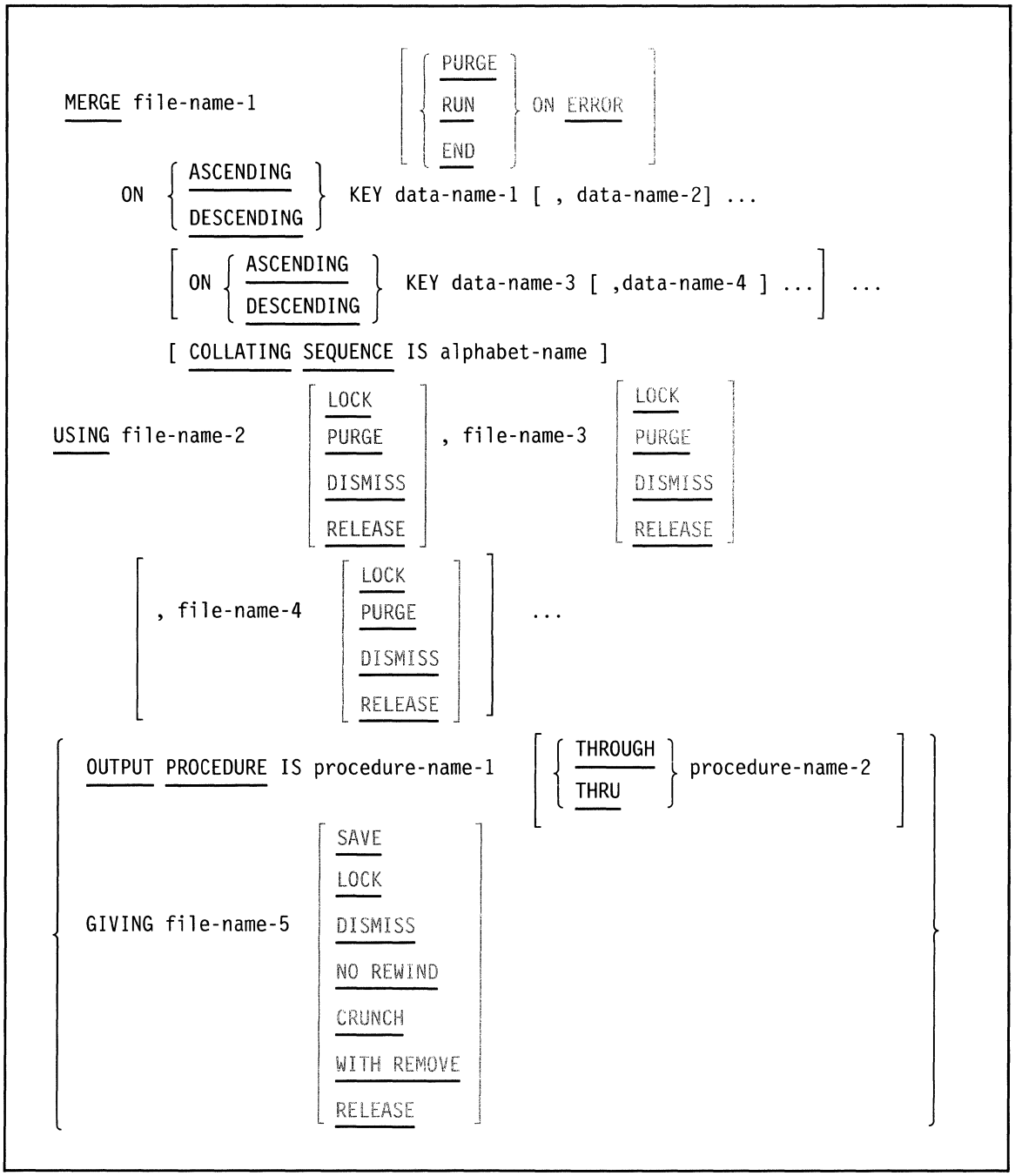
```
INSPECT word REPLACING CHARACTERS BY "B" BEFORE INITIAL "A".
```

Word before is 1 2 X Z A B C D
Word after is B B B B A B C D

MERGE Statement

The MERGE statement combines two or more identically sequenced files on a set of specified keys. During the process, it makes records available, in merge order, to an output procedure or to an output file.

The following is the general format:


MERGE file-name-1
[{ PURGE / RUN / END } ON ERROR]
ON { ASCENDING / DESCENDING } KEY data-name-1 [, data-name-2] ...
[ON { ASCENDING / DESCENDING } KEY data-name-3 [,data-name-4] ...] ...
[COLLATING SEQUENCE IS alphabet-name]
USING file-name-2 [LOCK / PURGE / DISMISS / RELEASE] , file-name-3 [LOCK / PURGE / DISMISS / RELEASE]
[, file-name-4 [LOCK / PURGE / DISMISS / RELEASE]] ...
{ OUTPUT PROCEDURE IS procedure-name-1 [{ THROUGH / THRU } procedure-name-2]
GIVING file-name-5 [SAVE / LOCK / DISMISS / NO REWIND / CRUNCH / WITH REMOVE / RELEASE] }

MERGE statements can appear anywhere except in the declarative portion of the Procedure Division or in an input or output procedure associated with a SORT or MERGE statement.

The MERGE statement merges all records contained in file-name-2, file-name-3, and file-name-4. The files referenced in the MERGE statement must not be open at the time the MERGE statement is executed. The merge operation automatically opens and closes these files, performing all implicit functions such as executing any associated USE procedures. Files are terminated as if a CLOSE statement, without optional phrases, had been executed for each file.

No more than one file-name from a multiple file reel can appear in the MERGE statement, and you must not repeat file-names within the MERGE statement.

You must describe file-name-1 in a sort-merge file description entry in the Data Division.

The ON ERROR option is provided to allow programmers some control over irrecoverable parity errors when INPUT/OUTPUT PROCEDURES are not present in a program. Using the PURGE option causes all records in a block containing an irrecoverable parity error to be dropped, and processing continues after a message appears on the ODT, giving the relative position of the bad block. RUN causes the program to use the bad block and provides the same message as that defined for PURGE. END causes the program to terminate. END is assumed if you do not specify an option.

You can specify as many as eight file-names in the USING clause.

Procedure-name-1 represents the name of an output procedure.

Describe file-name-2, file-name-3, file-name-4, and file-name-5 in a file description entry, not in a sort-merge file description entry, in the Data Division. The actual size of the logical record(s) described for file-name-2, file-name-3, file-name-4, and file-name-5 must be equal to the actual size of the logical record(s) described for file-name-1. If the data descriptions of the elementary items that make up these records are not identical, it is the programmer's responsibility to describe the corresponding records in such a manner as to cause an equal number of character positions to be allocated for the corresponding records.

Data-name-1, data-name-2, data-name-3, and data-name-4 are KEY data-names subject to the following rules:

- You must describe the data items identified by KEY data-names in records associated with file-name-1.
- You can qualify KEY data-names.
- The data items identified by KEY data-names must not be variable length items.

If file-name-1 has more than one record description, then the data items identified by KEY data-names can all be described within one record description or in any combination of record descriptions. It is not necessary to redescribe the KEY data-names in each record description. However, none of the data items identified by KEY data-names can be described by an entry that either contains an OCCURS clause or is subordinate to an entry that contains an OCCURS clause.

List the data-names following the word KEY from left to right in order of decreasing significance no matter how they are divided into KEY phrases. In the format, data-name-1 is the major key, data-name-2 is the next most significant key, and so on.

When the ASCENDING phrase is specified, the merged sequence is from the lowest value of the contents of the data items identified by the KEY data-names to the highest value, according to the rules for comparing operands in a relation condition.

When the DESCENDING phrase is specified, the merged sequence is from the highest value of the contents of the data items identified by the KEY data-names to the lowest value, according to the rules for comparing operands in a relation condition.

The OUTPUT PROCEDURE must consist of one or more paragraphs or sections that appear contiguously in a source program and are not a part of any other procedure. To make merged records available for processing, you must include the execution of at least one RETURN statement in the output procedure. Control must not be passed to the output procedure except when a related SORT or MERGE statement is being executed. The output procedure can consist of any procedures needed to select, modify, or copy the records that are being returned, one at a time in merged order, from file-name-1. The restrictions on the procedural statements within the output procedure are as follows:

- The output procedure must not contain any transfers of control to points outside the output procedure; ALTER, GO TO, and PERFORM statements in the output procedure cannot refer to procedure-names outside the output procedure. COBOL statements are allowed that cause an implied transfer of control to declaratives.
- The output procedures must not contain any SORT or MERGE statements.
- The remainder of the Procedure Division must not contain any transfers of control to points inside the output procedures; ALTER, GO TO, and PERFORM statements in the remainder of the Procedure Division cannot refer to procedure-names within the output procedures.

If an output procedure is specified, control passes to it when the MERGE statement is executed. The compiler inserts a return mechanism at the end of the last paragraph or section in the output procedure. When control passes the last statement in the output procedure, the return mechanism terminates the merge and then passes control to the next executable statement after the MERGE statement. Before entering the output procedure, the merge procedure reaches a point at which it can select the next record in merged order when requested. The RETURN statements in the output procedure are the requests for the next record.

If the GIVING phrase is specified, all the merged records in file-name-1 are automatically written on file-name-5 as the implied output procedure for this MERGE statement.

When the rules for the comparison of operands in a relation condition indicate that the contents of all the key data items of one data record are equal to the contents of the corresponding key data items of one or more other data records, these records are returned as follows:

- The order of the associated input files is followed as specified in the MERGE statement.
- All records associated with one input file are returned prior to the return of records from another input file.

When the records in the files referenced by file-name-2, file-name-3, and file-name-4 are not ordered as described in the ASCENDING or DESCENDING KEY clauses, the MERGE acts as previously described, with one exception. All improperly ordered data records are placed on the output file or returned to the output procedure immediately after they are read. As a result, the output from the MERGE statement is not in a strict ASCENDING or DESCENDING KEY order.

Use the LOCK, PURGE, DISMISS, and RELEASE options to specify the type of file close for the USING files, file-name-2, file-name-3, and file-name-4. Use the SAVE, LOCK, DISMISS, NO REWIND, WITH REMOVE, CRUNCH, and RELEASE options to specify the type of file close for the GIVING file, file-name-5 (see "CLOSE Statement" in this section).

MOVE Statement

The MOVE statement transfers data, according to the rules of editing, to one or more data areas.

The following are the general formats:

Format 1

```
MOVE { identifier-1 } TO identifier-2 [ , identifier-3] ...
     { literal      }
```

Format 2

```
MOVE { CORRESPONDING } identifier-1 TO identifier-2
     { CORR          }
```

Identifier-1 and literal represent the sending area; identifier-2 and identifier-3 represent the receiving area.

The data designated by the literal or identifier-1 is moved first to identifier-2 and then to identifier-3. The rules referring to identifier-2 also apply to the other receiving areas.

Any subscripting or indexing associated with identifier-2 is evaluated immediately before the data is moved to the respective data item. Any subscripting or indexing associated with identifier-1 is evaluated only once, immediately before data is moved to the first of the receiving operands.

For example, in the following statements, the result of the first statement is equivalent to the next three statements:

```
MOVE a (b) TO b, c (b)

MOVE a (b) TO temp
MOVE temp TO b
MOVE temp TO c (b)
```

In the example, *temp* is an intermediate data item provided by the compiler, as explained under "Intermediate Data Item" in this section.

When you use the CORRESPONDING phrase, both identifiers must be group items. In this case (Format 2), selected items within identifier-1 are moved to selected items within identifier-2 according to the rules given for the CORRESPONDING phrase earlier in this section. The results are the same as they would be if the user had referred to each pair of corresponding identifiers in separate MOVE statements.

Table 6–5 summarizes the legality of various types of MOVE statements. The following paragraphs elaborate further on this topic.

Table 6–5 . Validity of Moves between Various Categories of Data Items

	Category of Receiving Data Item		
Category of Sending Data Item	**Alphabetic**	**Alphanumeric-Edited Alphanumeric**	**Numeric Integer Numeric Noninteger Numeric-Edited**
Alphabetic	Yes/3	Yes/1	No/1
Alphanumeric	Yes/3	Yes/1	Yes/2
Alphanumeric-Edited	Yes/3	Yes/1	No/1
Numeric Integer	No/2	Yes/1	Yes/2
Numeric Noninteger	No/2	No/3	Yes/2
Numeric-Edited	No/2	Yes/1	No/1

Note: *The term "Yes" indicates that the move is legal, and the number following indicates the paragraph under MOVE Results, which explains the behavior of that move. The term "No" indicates that the move is not legal, and the number following indicates the paragraph under Illegal MOVES, which prohibits the move.*

Any MOVE in which the sending and receiving items are both elementary items is an elementary move. Every elementary item belongs to one of the following categories: numeric, alphabetic, alphanumeric, numeric-edited, alphanumeric-edited. These categories are described in the PICTURE clause. Numeric literals belong to the category numeric, and nonnumeric literals belong to the category alphanumeric. The figurative constant ZERO belongs to the category numeric. The figurative constant SPACE belongs to the category alphabetic. All other figurative constants belong to the category alphanumeric.

Illegal Moves

The following elementary moves are not allowed:

- The figurative constant SPACE, a numeric-edited, alphanumeric-edited, or alphabetic data item must not be moved to a numeric or numeric-edited data item.
- A numeric literal, the figurative constant ZERO, a numeric data item, or a numeric-edited data item must not be moved to an alphabetic data item.
- A noninteger numeric literal or a noninteger numeric data item must not be moved to an alphanumeric or alphanumeric edited data item.
- An index data item cannot appear as an operand of a MOVE statement (see "USAGE clause" in Section 5).

MOVE Results

All elementary moves other than those previously listed are legal. Any necessary conversion of data from one form of internal representation to another takes place during legal elementary moves, along with any editing specified for the receiving data item, as follows:

- When an alphanumeric-edited or alphanumeric item is a receiving item, alignment and any necessary space filling takes place as defined under "Data Alignment" in Section 5.
  - If the size of the sending item is greater than the size of the receiving item, the excess characters are truncated on the right after the receiving item is filled.
  - If the sending item is described as signed numeric, the operational sign is not moved; if the operational sign occupied a separate character position (see "SIGN Clause" in Section 5), that character is not moved and the size of the sending item is considered to be one less than its actual size (in terms of standard data format characters).
- When a numeric or numeric-edited item is the receiving item, alignment by decimal point and any necessary zero-filling takes place as defined under "Data Alignment" in Section 5, except where zeroes are replaced because of editing requirements.
  - When a signed numeric item is the receiving item, the sign of the sending item is placed in the receiving item (see "SIGN Clause" in Section 5). The representation of the sign is converted as necessary. If the sending item is unsigned, a positive sign is generated for the receiving item.
  - When an unsigned numeric item is the receiving item, the absolute value of the sending item is moved, and no operational sign is generated for the receiving item.
  - When a data item described as alphanumeric is the sending item, data is moved as if the sending item were described as an unsigned numeric integer.

- When a receiving field is described as alphanumeric, justification and any necessary space-filling takes place as defined under "Data Alignment" in Section 5. If the size of the sending item is greater than the size of the receiving item, the excess characters are truncated on the right after the receiving item is filled.

Any non-elementary MOVE statement is treated exactly as if it were an alphanumeric to alphanumeric elementary move, except that data is not converted from one form of internal representation to another. In such a MOVE, the receiving area is filled without consideration for the individual elementary or group items contained within either the sending or receiving area, except as noted in the OCCURS clause.

MULTIPLY Statement

The MULTIPLY statement causes numeric data items to be multiplied and sets the values of data items equal to the results. The following are the general formats:

Format 1

```
MULTIPLY { identifier-1 } BY identifier-2 [ROUNDED]
         { literal-1    }

  [,identifier-3 [ROUNDED]  ... [; ON SIZE ERROR imperative-statement]
```

Format 2

```
MULTIPLY { identifier-1 } BY { identifier-2 } GIVING identifier-3 [ROUNDED]
         { literal-1    }    { literal-2    }

    [ , identifier-4 [ROUNDED ]  ...  [ ; ON SIZE ERROR imperative-statement]
```

When Format 1 is used, the value of identifier-1 or literal-1 is multiplied by the value of identifier-2. The product replaces the value of the multiplier (identifier-2). The same is true for identifier-1 or literal-1 with identifier-3.

When Format 2 is used, the value of identifier-1 or literal-1 is multiplied by identifier-2 or literal-2 and the result is stored in identifier-3 and identifier-4, the objects of the GIVING clause.

Each identifier must refer to a numeric elementary item, except that in Format 2 each identifier following the word GIVING must refer to either an elementary numeric item or an elementary numeric-edited item.

Each literal must be a numeric literal.

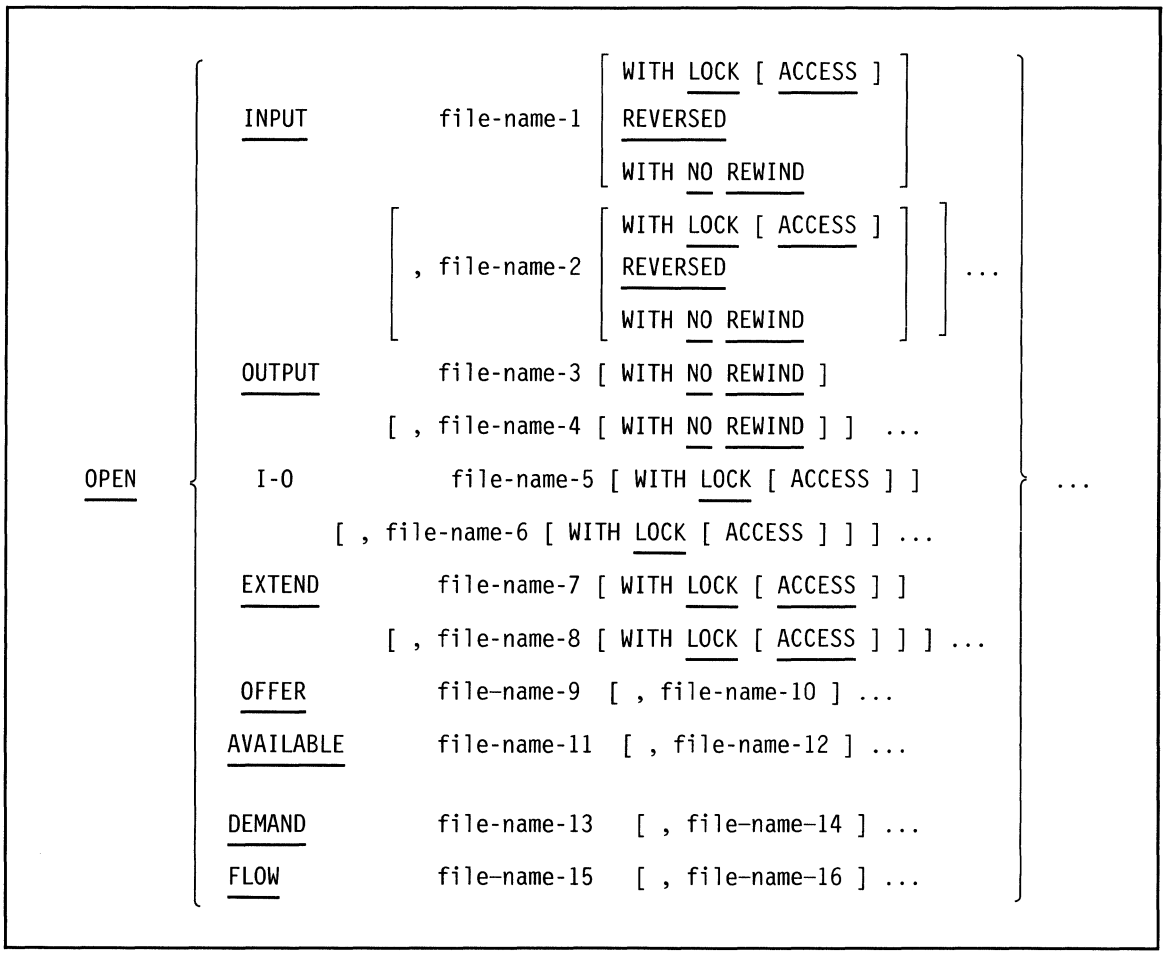
The composite of operands must not contain more than 98 digits. The composite of operands is the intermediate data item resulting from the superimposition of all receiving data items of a given statement aligned on their decimal points. Refer to "Intermediate Data Item" in this section.

Additional information can be found under "ROUNDED Phrase", "SIZE ERROR Phrase", "Arithmetic Statements", "Overlapping Operands", and "Multiple Results in Arithmetic Statements" in this section.

OPEN Statement

The OPEN statement initiates the processing of files. It also performs checking and/or writing of labels and other input/output operations.

The following is the general format:


OPEN
INPUT file-name-1 [WITH LOCK [ACCESS] | REVERSED | WITH NO REWIND]
[, file-name-2 [WITH LOCK [ACCESS] | REVERSED | WITH NO REWIND]] ...
OUTPUT file-name-3 [WITH NO REWIND]
[, file-name-4 [WITH NO REWIND]] ...
I-O file-name-5 [WITH LOCK [ACCESS]]
[, file-name-6 [WITH LOCK [ACCESS]]] ...
EXTEND file-name-7 [WITH LOCK [ACCESS]]
[, file-name-8 [WITH LOCK [ACCESS]]] ...
OFFER file-name-9 [, file-name-10] ...
AVAILABLE file-name-11 [, file-name-12] ...
DEMAND file-name-13 [, file-name-14] ...
FLOW file-name-15 [, file-name-16] ...
...

The successful execution of an OPEN statement opens a file after determining that it is available. Executing an OPEN statement does not affect either the contents or availability of the file's record area, nor does it obtain or release the first data record.

When a given file is not in an open mode, the only statement that can reference the file is a MERGE or a SORT statement with the USING or GIVING phrases.

You can open a file with the INPUT, OUTPUT, I-O, EXTEND, OFFER, and AVAILABLE phrases in the same program. Following the initial execution of an OPEN statement for a file, a CLOSE statement must precede any subsequent OPEN statement execution for that same file.

For files being opened with the INPUT or I-O phrase, the OPEN statement sets the current record pointer to the first record currently existing within the file. If no records exist in the file, the current record pointer is set so that the next executed READ statement for the file results in an AT END condition.

A file is created when an OPEN statement with the OUTPUT phrase is executed. At that time, the associated file contains no data records. Multiple files located on tape are allowed for sequential I/O. The VALUE OF FAMILYNAME clause specifies the corresponding family-id.

An OPEN statement must be successfully executed prior to executing any of the permissible input/output statements. In Table 6–6, an X at an intersection indicates that the specified statement, used in the access mode stated in the left column, can be used with the sequential file organization and open mode given at the top of the column.

Table 6–6. Permissible Statements

File Access Mode	Statement	Open Mode			
		Input	Output	I-O	Extend
Sequential	READ	X		X	
	WRITE		X		X
	REWRITE			X	
	START	X			X
	DELETE				X
Random	READ	X		X	
	WRITE		X	X	
	REWRITE			X	
	START				
	DELETE				X
Dynamic	READ	X		X	
	WRITE		X	X	
	REWRITE			X	
	START	X		X	
	DELETE			X	

The files referenced in the OPEN statement need not all have the same organization or access.

The file description entry for file-name-1, file-name-2, file-name-5, file-name-6, file-name-7, or file-name-8 must be equivalent to that used when this file was created.

Note: *Whenever a disk or diskpack file is opened input, the MCP reads up the AREAS and AREASIZE attributes as found in the disk or diskpack file header, not those declared in the program. These values as found in the file header are then written into the program FIB by the MCP and are not changed by a subsequent close and reopen OUTPUT of this file.*

You can use the I-O phrase only for mass storage files. The I-O phrase permits the opening of a mass storage file for both input and output operations. Since this phrase implies the existence of the file, you cannot use it when you first create the mass storage file.

The OPEN EXTEND statement allows you to write additional records to the end of a file. This construct is valid for all hardware device types.

When you use the EXTEND phrase, the OPEN statement positions the file immediately following the last logical record of that file. Subsequent WRITE statements referencing the file add records to the file as though the file had been opened with the OUTPUT phrase.

OPEN EXTEND requires that file-name-7 and file-name-8 be previously created files (already in the disk or diskpack directory or on tape).

The EXTEND option is valid only for sequential files. It must not be specified for multiple file tapes.

The EXTEND option causes the file to be opened with LOCK ACCESS.

The OFFER phrase applies to port files. The OFFER phrase specifies that the subfile can be offered for matching to another process and that control is returned immediately to the next statement without waiting for a match to occur.

The AVAILABLE phrase applies to port files. The AVAILABLE phrase specifies that a specific subfile is opened if it matches a port file that has already been offered. If a match does not occur, the subfile is not opened, and it is no longer considered for subsequent matching. Control is returned immediately to the next statement.

File-name-9, file-name-10, file-name-11, and file-name-12 must be port files. The following rules and conditions are relevant to port files:

- I-O offers the subfile for matching. The program is suspended until a match is found.
- OPEN I-O cannot have the WITH LOCK ACCESS clause.
- OFFER offers a subfile for matching. The program continues to run.
- AVAILABLE opens an offered subfile if it is available and has a match. The program continues to run.

The DEMAND phrase applies to Reader-Sorter files. The DEMAND phrase opens the file in demand mode.

The FLOW phrase applies to Reader-Sorter files. The FLOW phrase opens the file in flow mode.

OPEN DEMAND and OPEN FLOW initiate the input processing of the Reader-Sorter. At least one of the options must be specified before a file can be read.

When the DEMAND or FLOW phrase is specified, file-name-13 and file-name-15 must be files assigned to a Reader-Sorter.

If you designate an input file with the OPTIONAL phrase in its SELECT clause and the file is not present at the time the OPEN statement is executed, the operator is notified of this fact. At this time you can load the file or enter the MCP keyboard command OF. If the OF command is used, the first READ statement for this file causes the AT END condition to occur.

The LOCK and LOCK ACCESS options have meaning only for mass storage files; they are ignored for all other files.

OPEN WITH LOCK on a mass storage file denies the use of that file to all other programs in the mix. A program is suspended if it attempts to OPEN WITH LOCK a file that is already open to some program or if it attempts to open a file that is already OPEN WITH LOCK to some other program.

When an OPEN with the LOCK option (without ACCESS) is executed, the following occurs:

- If the specified file is already in an open mode, the program is suspended awaiting exclusive availability of the file.
- If the file is not currently in an open mode by any program, the file is opened.

OPEN WITH LOCK ACCESS on a mass storage file allows the subsequent opening of the file by other operating programs in the mix as long as they specifiy the open as OPEN INPUT. A program is suspended if it attempts to OPEN WITH LOCK ACCESS a file that is already OPEN I-O, or OPEN EXTEND to some program. A program is also suspended if it attempts to OPEN I-O or OPEN EXTEND a file that is already OPEN WITH LOCK ACCESS to some other program. When an OPEN WITH LOCK ACCESS is executed, the program is suspended if any of the following conditions exist:

- The file is currently OPEN WITH LOCK.
- The file is currently OPEN I-O or EXTEND.
- This OPEN is I-O or EXTEND and the file is currently in an OPEN INPUT LOCK ACCESS state.

The program remains suspended until none of these conditions holds or until the program terminates. In any other case, the OPEN WITH LOCK ACCESS causes the

file to be opened. A suspended program is resumed when the conflicting users of the file have closed the file.

When the REVERSED phrase is specified, the subsequent READ statements for the file make the data records of the file available in reversed order, that is, starting with the last record.

The REVERSED and the NO REWIND phrases can only be used with sequential single reel/unit files (see "CLOSE Statement" in this section).

The REVERSED and WITH NO REWIND phrases are ignored if they do not apply to the storage media on which the file resides.

If the storage medium for the file permits rewinding, the following rules apply:

- When neither the REVERSED, the EXTEND, nor the NO REWIND phrase is specified, executing the OPEN statement causes the file to be positioned at its beginning.
- When the NO REWIND phrase is specified, executing the OPEN statement does not cause the file to be repositioned; the file must already be positioned at its beginning prior to execution of the OPEN statement.
- When the REVERSED phrase is specified, the file is positioned at its end by executing the OPEN statement.

If label records are specified for the file, the beginning labels are processed as follows:

- When you specify the INPUT phrase, executing the OPEN statement causes the labels to be made available to the program.
- When you specify the OUTPUT phrase, executing the OPEN statement causes the labels to be written as specified in the program.
- For more information see "LABEL RECORDS Clause" in Section 5 and "USE Statement" in this section.

When you specify the EXTEND phrase and the LABEL RECORDS clause indicates label records are present, executing the OPEN statement includes the following steps:

- The beginning file labels are processed only in the case of a single reel/unit.
- The beginning reel/unit labels on the last existing reel/unit are processed as though the file were being opened with the INPUT phrase.
- The existing ending file labels are processed as though the file were being opened with the INPUT phrase. These labels are then deleted.
- Processing then proceeds as though the file had been opened with the OUTPUT phrase.

If label records are not specified, it is possible by operator intervention to equate the file to one with labels, in which case the label records are ignored.

PERFORM Statement

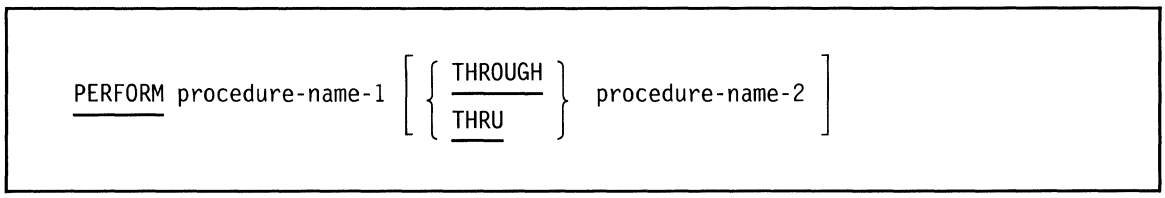
The function of this statement is to depart from the normal sequence of execution in order to execute one or more procedures, either a specified number of times or until a specified condition is satisfied.

The PERFORM statement transfers control explicitly to one or more procedures and returns control implicitly whenever execution of the specified procedure is complete.

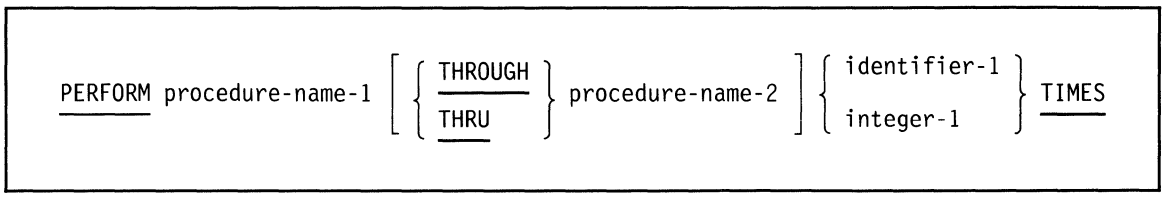
PERFORM is a powerful tool for abstraction and simplification in programs. Complex procedures can be kept in performed paragraphs, regardless of the number of places they are used. The main logic of a program can be easier to understand if it is kept simple, performing detailed operations rather than including them in-line.

The following are the general formats:

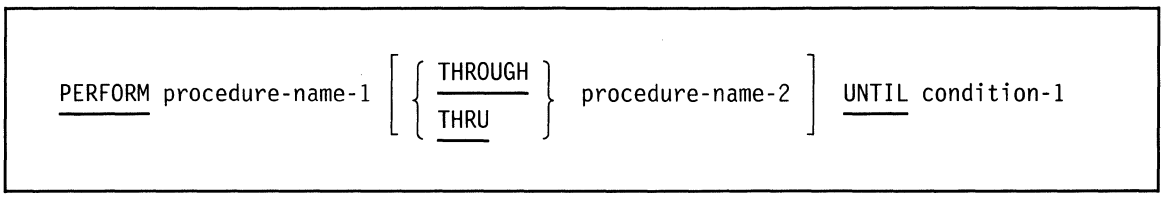
Format 1



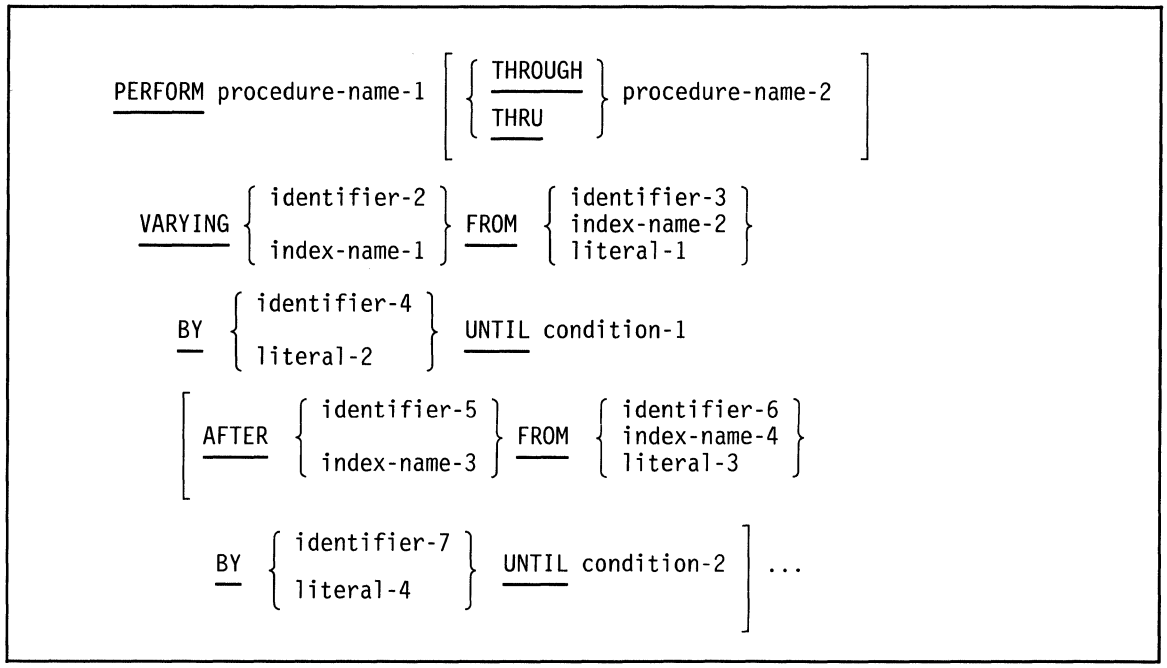
Format 2



Format 3



Format 4



THRU and THROUGH are equivalent.

PERFORM is the means by which subroutines are executed in COBOL. You can group sequences of statements that must be executed more than once into a paragraph and perform the paragraph rather than code the statements repeatedly at each place needed.

The subroutines can be executed any number of times, as determined by a variety of controls. A given paragraph can be performed by itself or in conjunction with another paragraph. Also, control can pass through it in sequential operation, and it can be the object of a GO statement all in the same program.

When you specify both procedure-name-1 and procedure-name-2 and either is the name of a procedure in the declarative section of the program, both must be procedure-names in the same declarative section.

Each identifier represents a numeric elementary item described in the Data Division. In Format 2, identifier-1 must be described as a numeric integer.

Each literal represents a numeric literal.

A data item or a literal in the BY phrase must not be zero.

Condition-1 and condition-2 can be any conditional expression, as described under "Conditional Expressions" in this section.

If an index-name is specified in the VARYING or AFTER phrase, then the following applies:

- The identifier in the associated FROM and BY phrases must be an integer data item.
- The literal in the associated FROM phrase must be a positive integer.
- The literal in the associated BY phrase must be a nonzero integer.

If an index-name is specified in the FROM phrase, then the following applies:

- The identifier in the associated VARYING or AFTER phrase must be an integer data item.
- The identifier in the associated BY phrase must be an integer data item.
- The literal in the associated BY phrase must be an integer.
- The identifier in the associated FROM phrase must have a positive value.

When the PERFORM statement is executed, control is transferred to the first statement of the procedure named procedure-name-1 (except as indicated under PERFORM Logic Cycle). This transfer of control occurs only once for each execution of a PERFORM statement. When the transfer does take place, an implicit transfer of control to the next executable statement following the PERFORM statement is established as follows:

- If procedure-name-1 is a paragraph-name and procedure-name-2 is not specified, then the return is after the last statement of procedure-name-1.
- If procedure-name-1 is a section-name and procedure-name-2 is not specified, then the return is after the last statement of the last paragraph in procedure-name-1.
- If procedure-name-2 is specified and it is a paragraph-name, then the return is after the last statement of the paragraph.
- If procedure-name-2 is specified and it is a section-name, then the return is after the last statement of the last paragraph in the section.

No particular sequential relationship is required between procedure-name-1 and procedure-name-2. There can be more than one logical path through the performed range of procedures. A common method of documenting the terminal paragraph in a range of procedures is through the use of the EXIT statement.

An implicit return mechanism is established at the beginning of performed procedures and is activated by executing a PERFORM statement. When reaching an active return mechanism, control returns to the associated PERFORM statement. A return mechanism permanently deactivates after transferring program control back to a PERFORM statement. A return mechanism is temporarily deactivated by executing a PERFORM statement. Program control always passes through a nonactive return mechanism to the executable statement following the PERFORM range.

A procedure executed under the control of a PERFORM statement can execute PERFORM statements. The range of procedures executed under the control of the nested PERFORM statement need not be declared totally within, or disjoint from, the range of procedures executed by the first PERFORM statement. Permanently deactivating an active return mechanism causes the last return mechanism temporarily deactivated to become active again, allowing overlapping PERFORM ranges, or two or more PERFORM ranges that have a common exit point, to logically execute the same as disjoint PERFORM ranges.

Using a GO statement to transfer program control from a range of procedures being executed under control of a PERFORM statement does not deactivate the return mechanism. Subsequently transferring program control back into the PERFORM range causes control to return to the PERFORM statement, provided that the return mechanism is still active. Repeatedly branching from a PERFORM range without ever allowing control to reach an active return mechanism can cause the program to terminate abnormally by exhausting the resources allocated to account for return mechanisms.

Note: *It is important that any performed procedures be exited via the normal return mechanism, as discussed above. Failure to do so can lead to program failure (usually with an address error) due to stack overflow.*

Each time a PERFORM is executed, an entry is made in the stack area of the program. That entry is not removed until the PERFORM ends with a normal return. If you execute a PERFORM, but the program then branches out of the performed routine and never returns to take the normal exit, the resulting stack entry is never removed. Repeated occurrences can fill all the area available for the stack.

Always use a PERFORM as intended: to execute a group of statements that happen to have been placed elsewhere in the program, then to continue to the next.

For a detailed discussion of program stacks, refer to the V Series MCP/VS Programming Reference Manual.

PERFORM Logic Cycle

The PERFORM statements operate as outlined in the following paragraphs.

Format 1

Format 1 is the basic PERFORM statement. A procedure referenced by this type of PERFORM statement is executed once. Then control passes to the next executable statement.

Format 2

Format 2 is the PERFORM...TIMES statement. The procedures are performed the number of times specified by integer-1 or by the initial value of the data item referenced by identifier-1 for that execution. If the value of the data item referenced by identifier-1 is less than or equal to zero, when the PERFORM statement is executed, control passes to the next executable statement. When the procedures have been executed the specified number of times, control is transferred to the next executable statement following the PERFORM statement.

When the PERFORM statement is executed, references to an identifier-1 cannot alter the number of times the procedures are to be executed from that indicated by the initial value of identifier-1.

Format 3

Format 3 is the PERFORM...UNTIL statement. The specified procedures are performed until the condition specified by the UNTIL phrase is true. When the condition is true, control is transferred to the next executable statement after the PERFORM statement.

If the condition is true when the PERFORM statement is entered, no transfer to procedure-name-1 takes place. Control is passed to the next executable statement following the PERFORM statement.

Format 4

Format 4 is the PERFORM...VARYING statement. This variation of the PERFORM statement is used to augment the values referenced by one or more identifiers or index-names in an orderly fashion during the execution of a PERFORM statement. Use the AFTER option when two or more conditions are required to control the number of times a procedure is performed. AFTER causes actions similar to VARYING but specifies that the actions of any preceding VARYING or AFTER clauses must not continue until the processing of the current AFTER clause has been processed. Up to 4 AFTER clauses are permitted. (See the example following the logic cycle flowcharts.)

In the following discussion, every reference to identifier as the object of the VARYING, AFTER, and FROM (current value) phrases also refers to index-names.

When index-name appears in a VARYING and/or AFTER phrase, it is initialized and subsequently augmented according to the rules of the SET statement.

When index-name appears in the FROM phrase, an identifier that appears in an associated VARYING or AFTER phrase (that is, identifier-2, identifier-5, and so forth) is initialized according to the rules of the SET statement; subsequent augmentation is as described in the following paragraphs.

Identifier-2 and all occurrences of identifier-5 are set in the order in which they occur in the PERFORM statement to the value of the literal or the current value of the identifier in the associated FROM phrase. Condition-1 is then evaluated.

If any condition, other than the last, is false on any occasion when it is evaluated, then the next condition is evaluated immediately.

If the last condition is false when evaluated, the sequence of procedures, procedure-name-1 through procedure-name-2, is executed once. The last BY value is then added to the associated identifier-2 or identifier-5, and the condition is evaluated again.

If condition-1 is true when evaluated, control is transferred to the next executable statement after the PERFORM statement. If condition-1 is true the first time it is evaluated, then the remaining conditions are not evaluated, and the sequence of procedures, procedure-name-1 through procedure-name-2, is not executed.

If any condition-2 is true when evaluated, then the associated identifier-5 is set to the value of literal-3 or the current value of identifier-6 in the associated FROM phrase. In addition, the identifier of the immediately preceding AFTER or VARYING phrase is incremented by the associated BY value, and the condition specified in that preceding phrase is then evaluated.

Figure 6–2 outlines the logic flow for the VARYING phrase of a PERFORM statement specifying only condition-1.

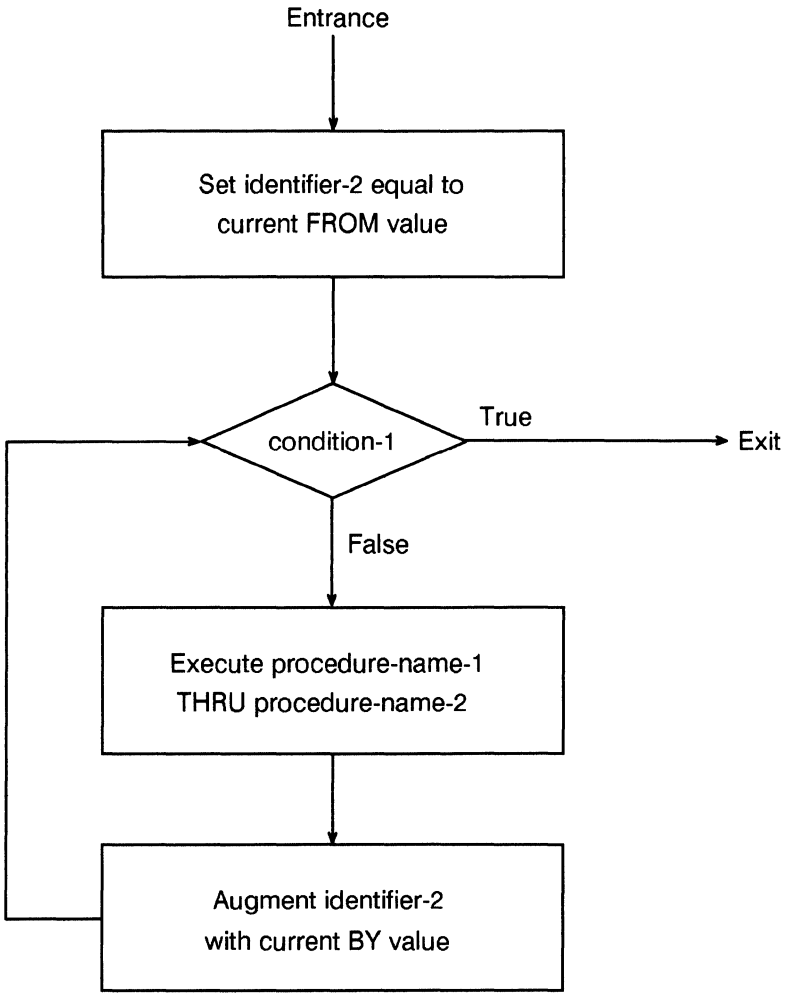


Figure 6–2. Flowchart for the VARYING Phrase of a PERFORM Statement Having One Condition.

For multiple identifiers, the mechanism is the same as it is for two identifiers, except that identifier-(3n-1) goes through a complete cycle each time that identifier-(3n-4) is augmented by the current BY value.

Figure 6–3 is a flowchart for the VARYING phrase of a PERFORM statement, which has *n* conditions.

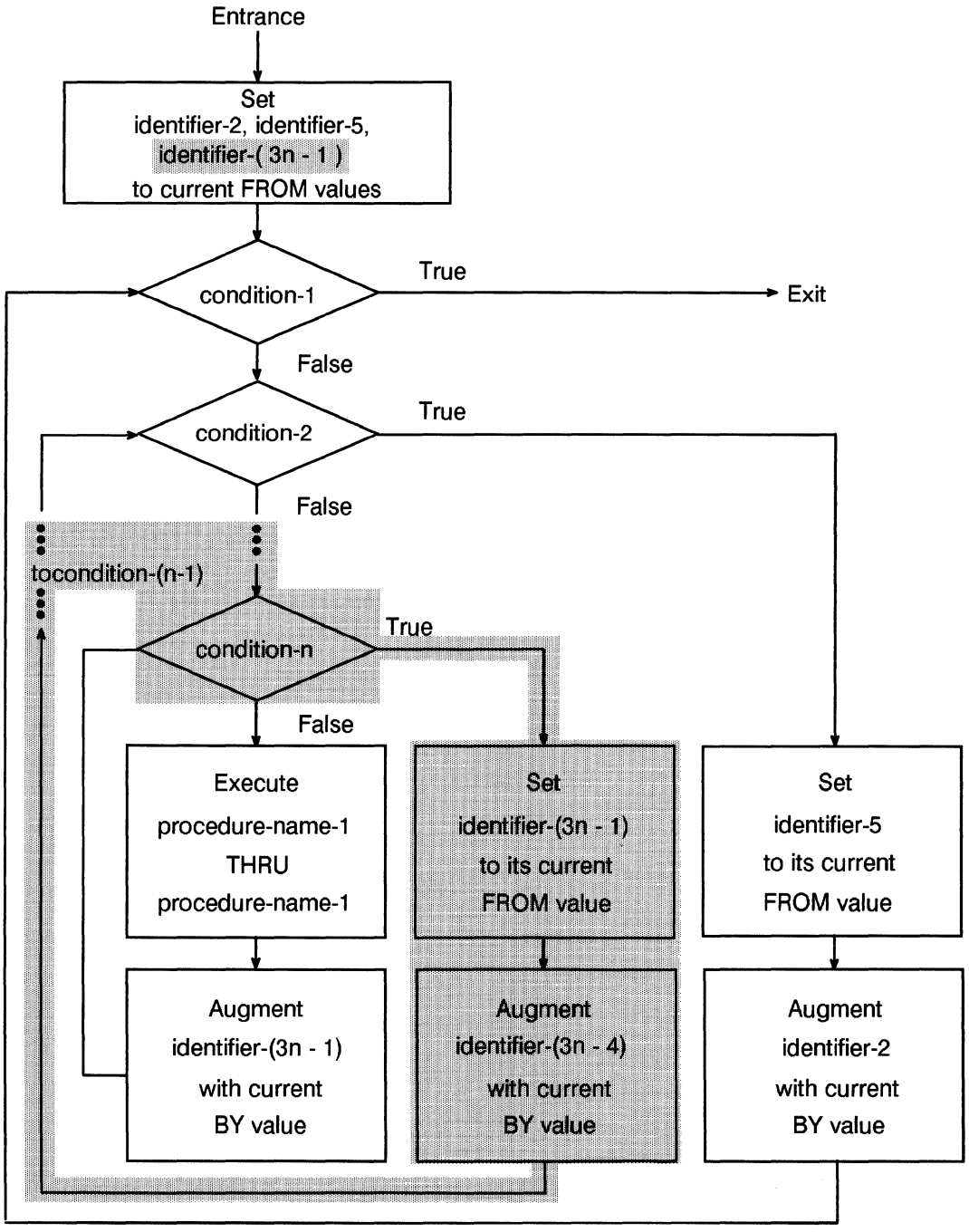


Figure 6–3. Flowchart for the VARYING Phrase of a PERFORM Statement Having More Than Two Conditions

The AFTER option is particularly useful when you are working with a multidimensional table. For example, assume the following table declaration:

```
01  TABLE-BODY.
    03  LEVEL-1  OCCURS 10 TIMES.
        05  LEVEL-2  OCCURS 10 TIMES.
            07  LEVEL-3  OCCURS 10 TIMES.
                09  TBL-ENTRY      PIC 9(4).
```

The following statements can access all elements in the table:

```
    PERFORM MODIFY-TABLE
        VARYING SUB1 FROM 1 BY 1
          UNTIL SUB1 IS GREATER THAN 10
        AFTER  SUB2 FROM 1 BY 1
          UNTIL SUB2 IS GREATER THAN 10
        AFTER  SUB3 FROM 1 BY 1
          UNTIL SUB3 IS GREATER THAN 10.
    .
    .
    .
MODIFY-TABLE.
    ADD CHANGE-FACTOR TO TBL-ENTRY (SUB1, SUB2, SUB3).
    .
    .
    .
```

In this example, the statements in the paragraph MODIFY-TABLE are executed once for each item in the table. The subscript SUB1 is assigned the values from 1 to 10. For each value of SUB1, SUB2 is assigned the values from 1 to 10. For each value of SUB2, SUB3 is assigned the values from 1 to 10. Each time a value is assigned to SUB3, MODIFY-TABLE is executed.

The item named in a VARYING or AFTER clause holds its current value until the action specified in the following AFTER clause is complete. That is, when SUB1 is assigned the value 1, it retains that value and SUB2 is assigned each value from 1 through 10. In turn, the value of SUB2 does not change until SUB3 has been assigned every value from 1 through 10. The first variable named is varied the least often; SUB1 is varied from 1 through 10 only once. The variable named in the final AFTER clause is varied the most often; SUB3 changes value 100 times for each value of SUB1.

READ Statement

For sequential access, the READ statement makes available the next logical record from a file. For random access, it makes available a specific record from a mass storage file. When a READ is executed, the record becomes available.

The following are the general formats:

Format 1

```
READ file-name [NEXT] RECORD [INTO identifier]
                             [; AT END {NEXT SENTENCE          }]
                                       {imperative-statement   }
```

Format 2

```
READ file-name RECORD [WITH NO WAIT] [INTO identifier] [; KEY IS data-name]
                      [; INVALID KEY  imperative-statement]
```

You must use format 1 for all files in sequential access mode. The NEXT phrase is applicable only to indexed or relative files; it is specifically for files in dynamic access mode when records are to be retrieved sequentially.

You must use format 2 for Indexed I/O files in random access mode or for files in dynamic access mode when records are to be retrieved randomly.

You must use the INVALID KEY phrase or the AT END phrase if no applicable USE procedure is specified for file-name.

The associated file must be open in the INPUT or I-O mode at the time this statement is executed. In format 1, the NEXT SENTENCE option can be specified in the AT END phrase; however, this option is valid only for sequential files. In the event of an end-of-file condition, control is transferred to the next executable statement after the READ.

Format 1

The record to be made available by the READ statement is determined as follows:

Sequential Files

If the current record pointer was positioned by executing an OPEN or SEEK statement, the record pointed to by the current record pointer is made available.

If the current record pointer was positioned by executing a previous READ statement, the current record pointer is updated to point to the next existing record in the file. Then that record is made available.

Relative Files

The record pointed to by the current record pointer is made available provided that the current record pointer was positioned by the START or OPEN statement and the record is still accessible through the path indicated by the current record pointer. If the record is no longer accessible, which could have been caused by the deletion of the record, the current record pointer is updated to point to the next existing record, which is then made available.

If the current record pointer was positioned by executing a previous READ statement, the current record pointer is updated to point to the next existing record in the file and then that record is made available.

Indexed Files

The record pointed to by the current record pointer is made available provided that the current record pointer was positioned by the START or OPEN statement and the record is still accessible through the path indicated by the current record pointer. If the record is no longer accessible, which could have been caused by the deletion of the record or a change in an alternate record key, the current record key is updated to point to the next existing record within the established key of reference and that record is then made available.

If the current record pointer was positioned by executing a previous READ statement, the current record pointer is updated to point to the next existing record in the file with the established key of reference and then that record is made available.

Format 1 Details

Executing the READ statement causes the value of the FILE STATUS data item, if any, associated with file-name to be updated (see "File-Control Paragraph" in Section 4).

When more than one record description describes the logical records of a file, these records automatically share the same storage area. This is equivalent to an implicit redefinition of the area. The contents of any data items that lie beyond the range of the current data record are undefined when the READ statement has been executed.

For a relative or indexed file for which dynamic access mode is specified, a Format 1 READ statement with the NEXT phrase specified causes the next logical record to be retrieved from that file, as described in the preceding paragraphs.

You can use the INTO phrase when the file input contains logical records of various sizes, as indicated by their record descriptions (that is, variable length records are allowed).

Using the INTO phrase moves the record being read from the record area to the area specified by identifier, according to the rules specified for the MOVE statement without the CORRESPONDING phrase. For this move, the sending area is treated as a group item whose size is equal to the maximum record size for this file. The implied move does not occur if executing the READ statement was unsuccessful. Any subscripting or indexing associated with identifier is evaluated after the record has been read and immediately before it is moved to the data item.

Using the INTO phrase makes the record being read available in both the input record area and the data area associated with identifier. The storage area associated with the identifier and the record area associated with the file-name must not be in the same storage area.

If the position of the current record pointer for that file is undefined, at the time of execution of a READ statement, executing that READ statement is unsuccessful.

If the end of a reel or unit is recognized during execution of a READ statement and the logical end of the file has not been reached, the following operations are executed:

- The standard ending reel/unit label procedure.
- A reel/unit swap.
- The standard beginning reel/unit label procedure.
- The first data record of the new reel/unit is made available.

If a file described with the OPTIONAL phrase is not present at the time the file is opened, then at the time of execution of the first READ statement for the file, the AT END condition occurs and executing the READ statement is unsuccessful. The standard end-of-file procedures are not performed. Execution of the program then proceeds with the actions taken when AT END is specified.

If, at the time of execution of a Format 1 READ statement, no next logical record exists in the file, the AT END condition occurs and executing the READ statement is unsuccessful.

When the AT END condition is recognized, the following actions occur in the following order:

1. If specified for this file, a value is placed into the FILE STATUS data item to indicate an AT END condition.
2. If the AT END phrase is specified in the statement causing the condition, control is transferred to the AT END imperative-statement. Any USE AFTER EXCEPTION procedure specified for this file is not executed.
3. If the AT END phrase is not specified, then a USE AFTER EXCEPTION procedure must be either explicitly or implicitly specified for this file, and that procedure is executed.

When the AT END condition occurs, execution of the input/output statement that caused the condition is unsuccessful.

Following the unsuccessful execution of any READ statement, the contents of the associated record area and the position of the current record pointer are undefined. For indexed files, the key of reference is also undefined.

When the AT END condition has been recognized, do not execute a Format 1 READ statement for that file without first executing one of the following:

- A successful CLOSE statement followed by a successful OPEN statement for that file
- A SEEK statement for a sequential file.
- A successful START statement for a relative or indexed file
- A successful Format 2 READ statement for that file

If the ACTUAL KEY phrase is specified for a file whose access mode is sequential, executing a Format 1 READ statement updates the contents of the ACTUAL KEY data item to the ordinal number of the logical record accessed.

If the RELATIVE KEY phrase is specified, executing a Format 1 READ statement updates the contents of the RELATIVE KEY data item so that it contains the relative record number of the record made available.

Format 2

Executing a Format 2 READ works as described in the following paragraphs.

Sequential Files

The Format 2 READ statement accesses the record specified by the contents of the ACTUAL KEY data item.

If the contents of the ACTUAL KEY data item are less than 1 or greater than the ordinal number of the last record ever written to the file when the read statement is executed, execution is unsuccessful and the INVALID KEY condition exists.

When a sequential file is accessed randomly, a physical read is performed only when the value of the ACTUAL KEY data item has changed since executing the last READ or WRITE. Thus, if this file is read with a particular ACTUAL KEY value, then the record is modified, and the file is read again with the same ACTUAL KEY value. The modified data is still present in the file's record area.

Relative Files

For a relative file, executing a Format 2 READ statement sets the current record pointer to and makes available the record whose relative record number is contained in the data item named in the RELATIVE KEY phrase for the file. If the file does not contain such a record, the INVALID KEY condition exists and execution of the READ statement is unsuccessful.

Indexed Files

If the KEY is not specified in a Format 2 READ statement for an indexed file, the prime record key is established as the key of reference for this retrieval. If the dynamic access mode is specified, this key of reference is also used for retrievals by any subsequent executions of Format 1 READ statements for the file until a different key of reference is established.

If the KEY phrase is specified in a Format 2 READ statement, data-name is established as the key of reference for this retrieval. If the dynamic access mode is specified, this key of reference is also used for retrievals by any subsequent executions of Format 1 READ statements for the file until a different key of reference is established.

Executing a Format 2 READ statement compares the value of the key of reference with the value contained in the corresponding data item of the stored records in the file until a record with an equal value is found. The current record pointer is positioned at this record, which is then made available. If no record is identified, the INVALID KEY condition exists and execution of the READ statement is unsuccessful.

For an indexed file being sequentially accessed, records with the same duplicate value in an alternate record key that is the key of reference are made available in the same order as that in which execution of WRITE or REWRITE statements (which create such duplicate values) release them.

Port Files

A READ statement causes the program to wait until a logical record is available. You can prevent this possibility for port files by specifying the WITH NO WAIT phrase. You can specify the WITH NO WAIT phrase only for port files.

A status key value of 94 indicates that no logical record was available for the read. If an ACTUAL KEY phrase is declared for a port file, the user is responsible for updating the phrase with an appropriate subfile index. If the ACTUAL KEY phrase is non-zero, the specified subfile is read. If the ACTUAL KEY is zero, a nonselective read procedure is performed, and the ACTUAL KEY is updated to indicate the subfile index of the subfile that was read.

For a nonselective read procedure, the first logical record to arrive at a subfile in the port is returned as the data for the READ statement. The operating system determines the subfile to be read and no specific selection algorithm is guaranteed. However, no subfile is read continuously at the expense of the other subfiles.

If no ACTUAL KEY is declared for the port file, the file must contain a single subfile, which is read.

DCP Files

The normal READ BCT is generated for the DCP function. You can declare a USE AFTER EXCEPTION routine in the DECLARATIVES section of the program to handle I/O errors. If the USE routine is specified, it is invoked when the MCP returns a type 29 result header for a READ. If you do not specify the USE routine, check for a 29 result header immediately after each READ of the DCP file.

RECEIVE Statement

The RECEIVE statement enables the user program to obtain data from another program in the same multiprogramming mix.

The following are the general formats:

Format 1

```
RECEIVE { identifier-1 } INTO identifier-2
        { literal-1    }

    [ON EXCEPTION imperative-statement]
```

Format 2

```
RECEIVE FROM { TOP    } identifier-3
             { BOTTOM }

  [ON EXCEPTION imperative-statement]
```

Format 1 is for synchronous communication between programs active in the same multiprogramming mix. It utilizes the core-to-core capabilities of the MCP. Format 2 is for asynchronous communication. The sending program need not be present in the multiprogramming mix at the same time as the receiving program executes the RECEIVE verb. Format 2 utilizes the MCP's Storage Queue capabilities.

Format 1

Identifier-1 and literal-1 must be nonnumeric data items specifying the file-id of the sending program. The sending program must be present in the MCP mix.

If the value of identifier-1 or literal-1 is all blanks, any number of sending programs become eligible for the transmission of data.

Identifier-2 must be an alphanumeric data item contained within the receiving program.

When the RECEIVE statement is executed, the EXCEPTION condition exists if the program named in identifier-1 or literal-1 is not ready to send or if identifier-1 or literal-1 contains all blanks and no program in the mix is ready to send. In that case

- If the ON EXCEPTION clause is specified, the imperative statement is executed.
- If the ON EXCEPTION clause is not specified, the program is suspended until the RECEIVE request has been filled.

The MCP's core-to-core (CRCR) option must be set in order to use this statement.

Format 2

Identifier-3 must refer to a level 01 data description entry that contains a description of a STOQUE parameter block. See the "SEND Statement" in this section for a complete description of the Storage Queue function and of the STOQUE parameter block. The following is the structure of a STOQUE parameter block:

```
01  Identifier-3.
    02  Queue-name              PIC X(6).
    02  Entry-name-length       PIC 9(2) COMP.
    02  Entry-name              PIC X(nn).
    02  Entry-data-length       PIC 9(4) COMP.
    02  Entry-data              PIC X(nnnn).
```

The message is received from the queue-name specified in the STOQUE parameter block named by identifier-3. If an entry-name is also specified, the message is received from the item or items (sub-queue) within queue-name that match the entry-name.

The message is placed in the entry-data field. Entry-data-length specifies the size of the message that was sent. If the message in the queue is longer than entry-data-length, only the first entry-data-length characters are received and the entry-data-length field is adjusted to show the actual number of characters of the message in the queue. If the message in the queue is shorter, the entry-data-length field is also adjusted to show the actual number of characters in the message.

Caution

Since this adjustment feature allows entry-data-length to be adjusted to a value larger than the actual length of entry-data, data corruption could result if the next message received is also longer than the length of entry-data. To avoid this situation, you should ensure that entry-data-length does not exceed the maximum value of entry-data prior to each receive.

More than one item in the queue can have the same name; the entry-name need not be unique. Also, the name given to an item when stored by the SEND verb can be longer than the name specified in the entry-name for a RECEIVE request. In either case, the queue is searched for an item whose name matches the entry-name in the first entry-name-length characters, as follows:

- If you specify RECEIVE FROM TOP, the first entry that is from the top of the queue and that meets the selection criteria is selected.
- If you specify RECEIVE FROM BOTTOM, the first entry that is from the bottom of the queue and that meets the selection criteria is selected.

When the request is complete, execution resumes at the next statement.

If the queue is empty or no individual entry satisfies the specified name, the ON EXCEPTION condition exists. In that case

- If you included the ON EXCEPTION clause, the imperative-statement is executed.
- If you did not include the ON EXCEPTION clause, the program is suspended until the requested item is placed into the queue.

To use this option of the RECEIVE statement, you must set the MCP's Storage Queue (STOQ) option.

RELEASE Statement

The RELEASE statement transfers records to the initial phase of a SORT operation.

The following is the general format:

```
RELEASE record-name [FROM identifier]
```

Executing a RELEASE statement releases the record identified by record-name to the initial phase of a sort operation. That is, a RELEASE statement causes a record to be written into the sort-merge file. Executing a RELEASE statement has no effect upon either the contents or accessibility of the record area.

A RELEASE statement can only be used within the range of an input procedure associated with a SORT statement for a file whose sort-merge file description entry contains record-name.

Record-name must be the name of a logical record in the associated sort-merge file description entry. It can be qualified.

Record-name and identifier must not refer to the same storage area.

If you use the FROM phrase, the contents of the identifier data area are moved to record-name. Then the contents of record-name are released to the sort file. Moving follows the rules specified for the MOVE statement without the CORRESPONDING phrase. If you do not use the FROM phrase, then you must use some other method to move data into the sort-merge file record area.

If you name the sort-merge file in a SAME RECORD AREA clause, the logical record is also available both to the program as a record of other files referenced in the same SAME RECORD AREA clause as references to the associated sort-merge file, and to the file associated with record-name. When control passes from the input procedure, the file consists of all those records that were placed in it by executing RELEASE statements.

RETURN Statement

The RETURN statement obtains either sorted records from the final phase of a SORT operation or merged records during a MERGE operation.

The following is the general format:

```
RETURN  file-name  RECORD  [INTO identifier] ;  AT END imperative statement
```

Executing the RETURN statement transfers to the record area the next record in the order specified by the KEY clause in the SORT or MERGE statement. That is, the RETURN statement causes the next record in the sort-merge file to be read.

A RETURN statement can only be used within the range of an output procedure associated with a SORT or MERGE statement for file-name.

File-name must be described by a sort-merge file description entry in the Data Division.

When the logical records of a file are described with more than one record description, these records automatically share the same storage area; this is equivalent to an implicit redefinition of the area. The contents of any data items that lie beyond the range of the current data record are undefined at the completion of executing the RETURN statement.

If records smaller than the maximum record size have been released, the contents of the record area beyond the end of the released record are unpredictable when subsequently returned. If necessary, you must account for the size of the logical record returned.

The INTO phrase can be used when the input file contains logical records of various sizes, as indicated by their record descriptions (that is, variable length records are allowed).

If the INTO phrase is specified, the current record is moved from the input area to the area specified by identifier, according to the rules for the MOVE statement without the CORRESPONDING phrase; the sending area is considered to be a group item having a fixed size equal to the maximum record size. The implied MOVE does not occur if there is an AT END condition. Any subscripting or indexing associated with identifier is evaluated after the record has been returned and immediately before it is moved to the data item.

When the INTO phrase is used, the data is available both in the input record area that is the sort-merge record area, and in the data area associated with identifier. The storage area associated with the identifier and the record area associated with the file-name must not be in the same storage area.

If no next logical record exists for the file when a RETURN statement is executed, the AT END condition occurs.

The contents of the record areas associated with the file when the AT END condition occurs are undefined.

When the AT END condition occurs, no transfer of data to the record area takes place and the contents of the record area are undisturbed. After the AT END condition has occurred, the contents of the record area are still accessible.

The NEXT SENTENCE phrase is valid only for sequential files, and in the event of an end-of-file condition, control is transferred to the next executable statement.

After executing the imperative-statement in the AT END phrase, you cannot execute the RETURN statement as part of the current output procedure.

REWRITE Statement

The REWRITE statement logically replaces a record existing in a mass storage file.

The following is the general format:

```
REWRITE record-name  [FROM identifier]
    [; INVALID KEY imperative-statement]
```

Record-name is the name of a logical record in the File Section of the Data Division. Record-name can be qualified.

Record-name and identifier must not refer to the same storage area.

The file associated with record-name must be a mass storage file open in the I/O mode at the time this statement is executed.

For files in the sequential access mode, the last input/output statement executed for the associated file prior to executing the REWRITE statement must have been a successfully executed READ statement. The REWRITE logically replaces the record that was accessed by the READ.

The number of character positions in the record referenced by record-name must be equal to the number of character positions in the record being replaced.

Executing the REWRITE statement has no effect upon the contents or accessibility of the record area. If the associated file is named in a SAME RECORD AREA clause, the logical record is available to the program as a record of other files appearing in the same SAME RECORD AREA clause as the associated I/O file appears, as well as to the file associated with record-name.

Executing a REWRITE statement with the FROM phrase is equivalent to executing *MOVE identifier TO record-name* followed by executing the same REWRITE statement without the FROM phrase. The contents of the record area prior to executing the implicit MOVE statement have no effect on executing the REWRITE statement.

The current record pointer is not affected by executing a REWRITE statement.

Executing the REWRITE statement causes the value of any FILE STATUS data item associated with the file to be updated (see "File-Control Paragraph" in Section 4).

The contents of the ACTUAL KEY data item, if specified, are ignored during the execution of the REWRITE statement and are not updated by execution of that statement.

For a relative file accessed in either random or dynamic access mode, the record specified by the contents of the RELATIVE KEY data item associated with the file is the record logically replaced. If the file does not contain the record specified by the key, then the INVALID KEY condition exists, the updating operation does not take place, and the data in the record area is unaffected.

The following applies to relative files:

- The INVALID KEY phrase must not be specified for a REWRITE statement that references a file in the sequential access mode.
- The INVALID KEY phrase must be specified in the REWRITE statement for files in the random or dynamic access mode for which an appropriate USE procedure is not specified.

The following applies to indexed files:

- For a file in the sequential access mode, the record to be replaced is specified by the value contained in the prime record key. When the REWRITE statement is executed, the value contained in the prime record key data item of the record to be replaced must be equal to the value of the prime record key of the last record read from this file.
- For a file in the random or dynamic access mode, the record to be replaced is specified by the prime record key data item.
- The contents of alternate record key data items of the record being rewritten could differ from those in the record being replaced. In this case, the alternate record key references to the record being rewritten are replaced by the new alternate record key references to the record being replaced.
- The INVALID KEY condition exists when one or more of the following are true:
  - The access mode is sequential and the value contained in the prime record key data item of the record to be replaced is not equal to the value of the prime record key of the last record read from the file.
  - The value contained in the prime record key data item is not equal to that of any record stored in the file.
  - The value contained in an alternate record key data item for which a DUPLICATES clause has not been specified is equal to that of a record already stored in the file.

  When an INVALID KEY condition occurs, the updating operation does not take place and the data in the record area is unaffected.

SEARCH Statement

The SEARCH statement searches a table for a table element that satisfies the specified condition and adjusts the associated index-name to indicate that table element.

The following are the general formats:

Format 1

```
SEARCH identifier-1 [ VARYING { identifier-2 } ]
                              { index-name-1 }

     [; AT END imperative-statement-1]

      ; WHEN condition-1 { imperative-statement-2 }
                         { NEXT SENTENCE          }

     [; WHEN condition-2 { imperative-statement-3 } ]
                         { NEXT SENTENCE          }
```

Format 2

```
SEARCH ALL identifier-1 [; AT END imperative-statement-1]

; WHEN { data-name-1      { IS EQUAL TO } { identifier-3           } }
     {                    { IS =        } { literal-1              } }
     { condition-name-1                   { arithmetic-expression-1} }

[  AND { data-name-2      { IS EQUAL TO } { identifier-4           } } ]
       {                  { IS =        } { literal-2              } }  ...
       { condition-name-2                 { arithmetic-expression-2} }

{ imperative-statement-2 }
{ NEXT SENTENCE          }
```

Note: *The required relational character = is not underlined in these syntax diagrams to avoid confusion with other symbols.*

In both Formats 1 and 2, identifier-1 must not be subscripted or indexed, but its description must contain an OCCURS clause and an INDEXED BY clause. The description of identifier-1 in Format 2 must also contain the KEY IS phrase in its OCCURS clause.

If identifier-1 is a data item subordinate to a data item that contains an OCCURS clause (providing for a multidimensional table), an index-name must be associated with each dimension of the table through the INDEXED BY phrase of the OCCURS clause. Only the setting of the index-name associated with identifier-1 (and the data item identifier-2 or index-name-1, if present) is modified by executing the SEARCH statement. To search an entire multidimensional table, you must execute a SEARCH statement several times. Prior to each execution of a SEARCH statement, you must execute SET statements whenever index-names must be adjusted to appropriate settings.

Refer to "Tables" in Section 5 for a complete discussion of tables, subscripting, and indexing.

After execution of every imperative-statement-1, imperative-statement-2, or imperative-statement-3 that does not terminate with a GO TO statement, control passes to the next executable sentence.

Format 1

Condition-1, condition-2, and so on can be any condition, as described under "Conditional Expressions," earlier in this section.

If the VARYING phrase is not used, the index-name that is used for the search operation is the first (or only) index-name that appears in the INDEXED BY phrase of identifier-1. Any other index-names for identifier-1 remain unchanged.

If you specify the VARYING index-name-1 phrase and index-name-1 appears in the INDEXED BY phrase of identifier-1, that index-name is used for this search. If this is not the case or you do not specify the VARYING identifier-2 phrase, the first (or only) index-name given in the INDEXED BY phrase of identifier-1 is used for the search. In addition, the following operations occur:

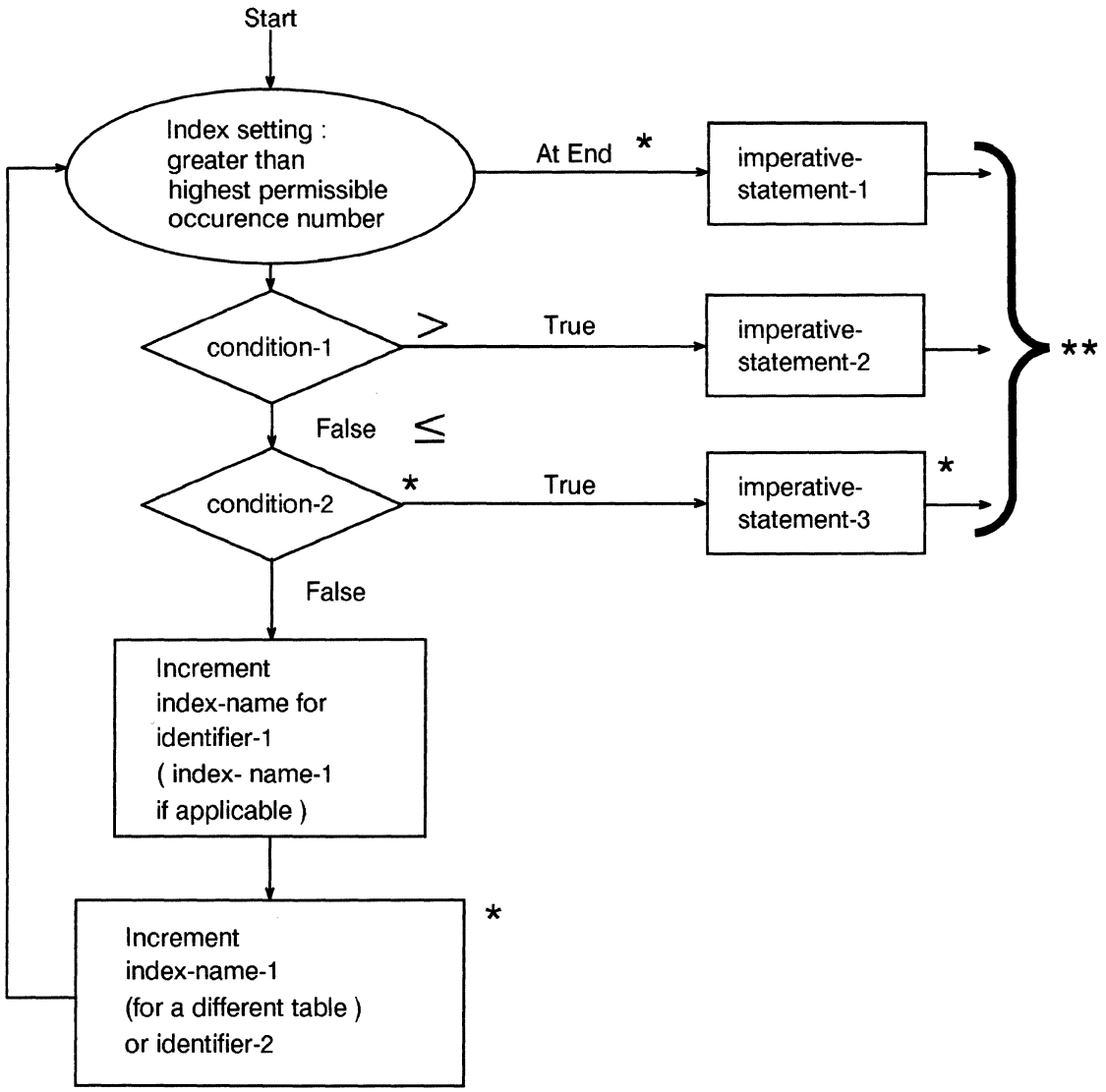
- If the VARYING index-name-1 phrase is used and index-name-1 appears in the INDEXED BY phrase of another table entry, the occurrence number represented by index-name-1 is incremented by the same amount as, and at the same time as, the occurrence number represented by the index-name associated with identifier-1.
- If the VARYING identifier-2 phrase is specified and identifier-2 is an index data item, then the data item referenced by identifier-2 is incremented by the same amount as, and at the same time as, the index associated with identifier-1. If identifier-2 is not an index data item, the data item referenced by identifier-2 is incremented by the value 1 at the same time as the index referenced by the index-name associated with identifier-1 is incremented.

Identifier-2, when specified, must be described as USAGE IS INDEX or as a numeric elementary item without any positions to the right of the assumed decimal point.

Format 1 of the SEARCH statement generates a serial type of search operation, starting with the current index setting.

- If, at the start of execution of the SEARCH statement, the index-name associated with identifier-1 contains a value that corresponds to an occurrence number that is greater than the highest permissible occurrence number for identifier-1, the SEARCH ends immediately. The number of occurrences of identifier-1, the last of which is the highest permissible, is discussed in the OCCURS clause. Then, if the AT END phrase is specified, imperative-statement-1 is executed. If the AT END phrase is not specified, control passes to the next executable sentence.
- If, at the start of execution of the SEARCH statement, the index-name associated with identifier-1 contains a value that corresponds to an occurrence number that is not greater than the highest permissible occurrence number for identifier-1 (the number of occurrences of identifier-1, the last of which is the highest permissible, is discussed under the "OCCURS Clause" in Section 5), the SEARCH statement evaluates the conditions in the order in which they are written, using the index settings, wherever specified, to determine the occurrence of those items to be tested. If none of the conditions are satisfied, the index-name for identifier-1 is incremented to obtain reference to the next occurrence. The process is then repeated using the new index-name settings, unless the new value of the index-name settings for identifier-1 corresponds to a table element outside the permissible range of occurrence values (greater than highest permissible occurence number). In this case, the search terminates as indicated in the preceding paragraph. If one condition is satisfied, the search terminates immediately and the imperative statement associated with that WHEN condition is executed; the index-name remains set at the occurrence that caused the condition to be satisfied.

Figure 6–4 is a flowchart depicting a Format 1 SEARCH operation containing two WHEN phrases.



* These options are included only when specified in the search statement.

** Each control transfer goes to the next executable sentence unless the imperative statement ends with a GO -TO statement.

Figure 6–4. Flowchart for a Format 1 SEARCH Statement Containing Two WHEN Phrases

Format 2

The index-name that is used for the SEARCH ALL operation is the first (or only) index-name that appears in the INDEXED BY phrase of identifier-1. Any other index-names for identifier-1 remain unchanged.

All referenced condition-names must be defined as having only a single value. The data-name associated with a condition-name must appear in the KEY clause of identifier-1. Data-name-1 and data-name-2 can be qualified. Each data-name-1, data-name-2 must be indexed by the first index-name associated with identifier-1, along with other indices or literals as required, and must be referenced in the KEY clause of identifier-1. Identifier-3, identifier-4, or identifiers specified in arithmetic-expression-1, arithmetic-expression-2 must not be referenced in the KEY clause of identifier-1 or be indexed by the first index-name associated with identifier-1.

When a data-name in the KEY clause of identifier-1 is referenced or when a condition-name associated with a data-name in the KEY clause of identifier-1 is referenced, all preceding data-names in the KEY clause of identifier-1 or their associated condition-names must also be referenced.

In a Format 2 SEARCH, the results of the SEARCH ALL operation are predictable only when the data in the table is ordered in the manner described in the ASCENDING/DESCENDING KEY clause associated with the description of identifier-1.

If the SEARCH ALL statement specifies only one WHEN condition and this condition is an EQUAL condition, a hardware SEA operation code is generated. Otherwise, a binary search takes place. The initial setting of the index-name for identifier-1 is ignored and its setting is varied during the search operation. At no time is the index name set to a value that exceeds the value corresponding to the last element of the table, or that is less than the value corresponding to the first element of the table. The length of the table is discussed under the "OCCURS clause" in Section 5.

If any of the conditions specified in the WHEN clause cannot be satisfied for any setting of the index within the permitted range, control passes to imperative-statement-1 of the AT END phrase when specified, or to the next executable sentence when this phrase is not specified. In either case, the final setting of the index is the value corresponding to an occurrence number that is one greater than the last element of the table.

If all the conditions can be satisfied, the index indicates an occurrence that allows the conditions to be satisfied, and control passes to imperative-statement-2.

When the contents of the data-name(s) referenced in the WHEN clause are not sufficient to identify a unique table element, the index indicates the occurrence closest to the beginning of the table that satisfies the condition.

SEEK Statement

The SEEK statement initiates the accessing of a mass storage data record for subsequent reading or writing.

The following is the general format:

```
SEEK file-name RECORD
```

The SEEK statement can serve to prelocate the address of a record for a mass storage file to facilitate the overlap of I/O and processor operations.

File-name must be the name of a mass storage file with SEQUENTIAL organization.

The ACTUAL KEY clause must be specified in the FILE-CONTROL entry for file-name. The value of the ACTUAL KEY specified for file-name is used as the ordinal number of the record sought.

For OUTPUT files, the ACCESS MODE should be RANDOM. Otherwise, the end-of-file pointer is changed after executing the SEEK statement.

For files specified for SEQUENTIAL access, the SEEK statement serves to reposition the file so that if the next statement that references the file is a READ statement, the record accessed is that specified by the contents of the ACTUAL KEY data item when the SEEK statement is executed.

Executing a SEEK statement does not cause the contents of the FILE STATUS data item to be updated.

SEND Statement

The SEND statement enables the user program to converse from a self-contained data-item with another program operating in the same multiprogramming mix.

The following are the general formats:

Format 1

```
SEND { identifier-1 } FROM identifier-2
     { literal-1    }

     [ON EXCEPTION imperative-statement]
```

Format 2

```
SEND TO { TOP    } identifier-3
        { BOTTOM }

        [ ON EXCEPTION  imperative-statement ]
```

Format 1 is for synchronous communication between programs active in the same multiprogramming mix, utilizing the MCP's core-to-core capabilities. Format 2 is for asynchronous communication; the receiving program need not be present in the multiprogramming mix at the same time as when the sending program executes the SEND statement. Format 2 utilizes the MCP's storage queue capabilities.

Format 1

Identifier-1 and literal-1 must be nonnumeric data items specifying the file-id of the receiving program. The receiving program must be present in the MCP mix. If the value of identifier-1 or literal-1 is all blanks, any number of receiving programs become eligible for the transmission of data.

The size of identifier-1 is limited only by the amount of memory required by the sending and receiving programs. Identifier-2 must be an alphanumeric data item contained within the sending program.

If the program named in identifier-1 or literal-1 is not ready to receive or if identifier-1 or literal-1 contains all blanks and no program in the mix is ready to

receive when the SEND statement is executed, then the EXCEPTION condition exists. In that case, the following applies:

- If the ON EXCEPTION clause is specified, the imperative statement is executed.
- If the ON EXCEPTION clause is not specified, the program is suspended until the SEND request has been filled.

The MCP's core-to-core (CRCR) option must be set if you want to use this statement.

Format 2

Identifier-3 must refer to a level 01 data description entry that contains a description of a STOQUE parameter block. The following is the structure of a STOQUE parameter block:

```
01  IDENTIFIER-3.
    02  QUEUE-NAME                    PIC X(6).
    02  ENTRY-NAME-LENGTH             PIC 9(2) COMP.
    02  ENTRY-NAME                    PIC X(nn).
    02  ENTRY-DATA-LENGTH             PIC 9(4) COMP.
    02  ENTRY-DATA                    PIC X(nnnn).
```

The following describes the elements of a STOQUE parameter block.

- Queue-name identifies the programmatically assigned symbolic name of the queue list to which the request pertains.
- Entry-name-length specifies the size of the optional entry name field.
- Entry-name, if present, specifies the name associated with the individual queue entry. This name can be used to provide a substructure to a list. The name provides the means to access data elements that are at locations other than the top or bottom of the queue.
- Entry-data-length specifies the size of the entry data area that in turn contains the transaction to be accessed for a storage request. The maximum size of the entry data field is 9999 bytes. This field serves as the response area for a queue inquiry request (ACCEPT MESSAGE COUNT).
- Entry-data, if present, contains the data to be added to the queue in a SEND operation. It receives the data in a RECEIVE operation. This field is not applicable to a queue inquiry request (ACCEPT MESSAGE COUNT) operation.

The MCP maintains a predefined number of queues in main storage, using the storage queue function. Data is added to a queue with the SEND verb and retrieved with the RECEIVE verb.

The message is sent from the entry-data field in the STOQUE parameter block specified by identifier-3. It is placed into the queue named in the queue-name field in the STOQUE parameter block. If an entry-name is also specified, the

message is identified by that entry-name within the queue. In this way, one or more sub-queues of messages can be assembled within the queue.

The message is sent from the entry-data field. Entry-data-length specifies the size of the message to be sent, in characters.

More than one item in the queue can have the same name; entry-name need not be unique. Also, the name given to an item when it is stored by the SEND verb can be longer than the name specified in entry-name for a RECEIVE request.

- If SEND TO TOP is specified, the item in the entry-data field is stored at the beginning of the queue named in the queue-name field in the STOQUE parameter block.
- If SEND TO BOTTOM is specified, the item is stored at the end of the queue named in the queue-name field of the STOQUE parameter block.

When the request is complete, execution resumes at the next statement.

If insufficient space exists in the queue for the storage request, the ON EXCEPTION condition exists. In that case the following applies:

- If the ON EXCEPTION clause is specified, the imperative-statement is executed.
- If the ON EXCEPTION clause is not specified, the program is suspended until the specified item is placed into the queue.

If a queue with the name given in the queue-name field does not already exist when the SEND statement is executed, the queue is created.

To use this form of the SEND statement, the MCP's storage queue (STOQ) option must be set.

Storage Queue Description

The STOQUE extension module of the MCP performs storage queue (STOQUE) functions. If its facilities are used, you must load it into memory by setting the STOQ system option.

The STOQUE module has the basic function of transferring data from a data area in the program to an external memory buffer and retrieving that data upon request. The mechanism can be used simultaneously by any number of programs as a means to transfer data between processes or even as temporary storage for a single process. The data elements placed into the memory buffer are organized into one or more program-independent, symbolically named lists called storage queues.

The STOQUE mechanism does not require synchronization of sending and receiving programs, as does the core-to-core function, since the STOQUE mechanism queues messages until they are requested. Multiple transactions can be in the storage queue simultaneously. Thus, using STOQUE permits processing

between programs to completely overlap, with no necessity to interlock for each transaction.

All requests made of the STOQUE function refer to the STOQUE parameter block in the user program, named in identifier-3 preceding. This program-maintained area contains the information needed by STOQUE to control the data elements; its format appears in the preceding section.

Programs can execute three types of calls on the STOQUE module: store data, retrieve data, and queue inquiry.

The storage constructs SEND TO TOP and SEND TO BOTTOM put data into the queue at the top or bottom, respectively. The first SEND executed causes the creation of a queue with the name specified if the queue does not already exist. If insufficient space is available in the queue for the storage request, the sending program is suspended until space becomes available unless the optional ON EXCEPTION clause has been specified.

The retrieval constructs RECEIVE FROM TOP and RECEIVE FROM BOTTOM retrieve data from the top or bottom of the queue, respectively. If the designated queue is empty or no individual entry satisfies any specified name constraint, the program is suspended until the desired element is placed into the queue unless the optional ON EXCEPTION clause has been specified.

The storage and retrieval functions have these important variations:

- Elements can be added to or removed from a list at either the top or the bottom, yielding the benefits of both FIFO (first-in-first-out) and LIFO (last-in-first-out) mechanisms.
- The elements in a queue list can be named, using entry-name and entry-name-length. In that case, access is first to the queue named in queue-name, and then to the specific element named in entry-name.

  The entry-name need not be unique; several elements in the queue might have the same name. The name given to a queue element when it is stored can be longer than the entry-name specified in a retrieval request. The queue is then searched for an element whose name, in the first entry-name-length characters, matches the specified entry-name mask. In either case of duplicate element names in a queue, the element accessed is the first matching name from the top of the queue on a RECEIVE FROM TOP or the first match from the bottom of the queue on a RECEIVE FROM BOTTOM.

The queue inquiry function provides a rapid means of determining the size of a list without disturbing any elements. The inquiry construct, ACCEPT MESSAGE COUNT, returns a count of the number of entries in a queue as an unsigned integer in the entry-data-length field. If an entry-name is specified, the count is the number of entries for that name or name group only. If the entry-name is omitted, the count is the total number of entries in the queue. A response of zero means that the queue or the designated portion of the queue is empty.

For further information on storage queue and core-to-core operations, see the *V Series MCP/VS Programming Reference Manual.*

SET Statement

The SET statement establishes reference points for table operations by setting index-names associated with table elements.

The following are the general formats:

Format 1

```
SET { identifier-1 } [ ,identifier-2 ] ... TO { identifier-3 }
    { index-name-1 }  [ ,index-name-2 ]       { index-name-3 }
                                              { integer-1    }
```

Format 2

```
SET index-name-4 [, index-name-5] ... { UP BY   } { identifier-4 }
                                      { DOWN BY } { integer-2    }
```

Index-names are related to a given table. They are defined by being specified in the INDEXED BY clause. Refer to "Tables" in Section 5 for a complete discussion of tables, subscripting, and indexing.

Identifier-1 and identifier-3 must name either index data items or elementary items described as an integer. Identifier-4 must be described as an elementary numeric integer. Integer-1 and integer-2 can be signed. Integer-1 can be zero.

If index-name-1 is specified, the value of the index after executing the SET statement should correspond to an occurrence number of an element in the associated table. If index-name-3 is specified, the value of the index before executing the SET statement should correspond to an occurrence number of an element in the associated table. If index-name-4 is specified, the value of the index both before and after executing the SET statement should correspond to an occurrence number of an element in the associated table.

All references to index-name-1, identifier-1, and index-name-4 apply equally to index-name-2, identifier-2, and index-name-5 respectively.

You can set index-name-1, index-name-2, index-name-4, and index-name-5 to any value, with the following restrictions:

- If overflow occurs, the value in the index-name is left truncated according to the arithmetic rules for a size error condition without a SIZE ERROR phrase.

- When you execute a statement using the index-name in reference to a table element, the value contained in the index or produced by relative indexing must fall within the range specified by the OCCURS clause defining the table. Otherwise, an abnormal termination of the program can occur, unless the dollar option NOBOUNDS was specified.

Table 6–7 shows the validity of various operand combinations using Format 1 of the SET statement. The numbers 1, 2, and 3 in the table refer to the paragraphs that follow, explaining the "Valid" or "No" entries in each column.

Table 6–7. Valid Operand Combinations for the SET...TO Statement

Sending Item	Receiving Item		
	Integer Data Item	**Index-name**	**Index Data Item**
Integer Literal	No[3]	Valid[1]	No[2]
Integer Data Item	No[3]	Valid[1]	No[2]
Index-Name	Valid[3]	Valid[1]	Valid[2, 4]
Index Data Item	No[3]	Valid[1, 4]	Valid[2, 4]

1 Index-name-1 is set to a value that causes it to refer to the table element that corresponds in occurrence number to the table element referenced by index-name-3, identifier-3, or integer-1. If identifier-3 is an index data item or if index-name-3 is related to the same table as index-name-1, no conversion takes place.

2 If identifier-1 is an index data item, it can be set equal to either the contents of index-name-3 or identifier-3, where identifier-3 is also an index data item. No conversion takes place in either case.

3 If identifier-1 is not an index data item, it can be set only to an occurrence number that corresponds to the value of index-name-3. Neither identifier-3 nor integer-1 can be used in this case.

4 No conversion takes place.

The process is repeated for index-name-2 and identifier-2, if specified. Each time the value of index-name-3 or identifier-3 is used as it was at the beginning of the execution of the statement. Any subscripting or indexing associated with identifier-1, and so on, is evaluated immediately before the value of the respective data item changes.

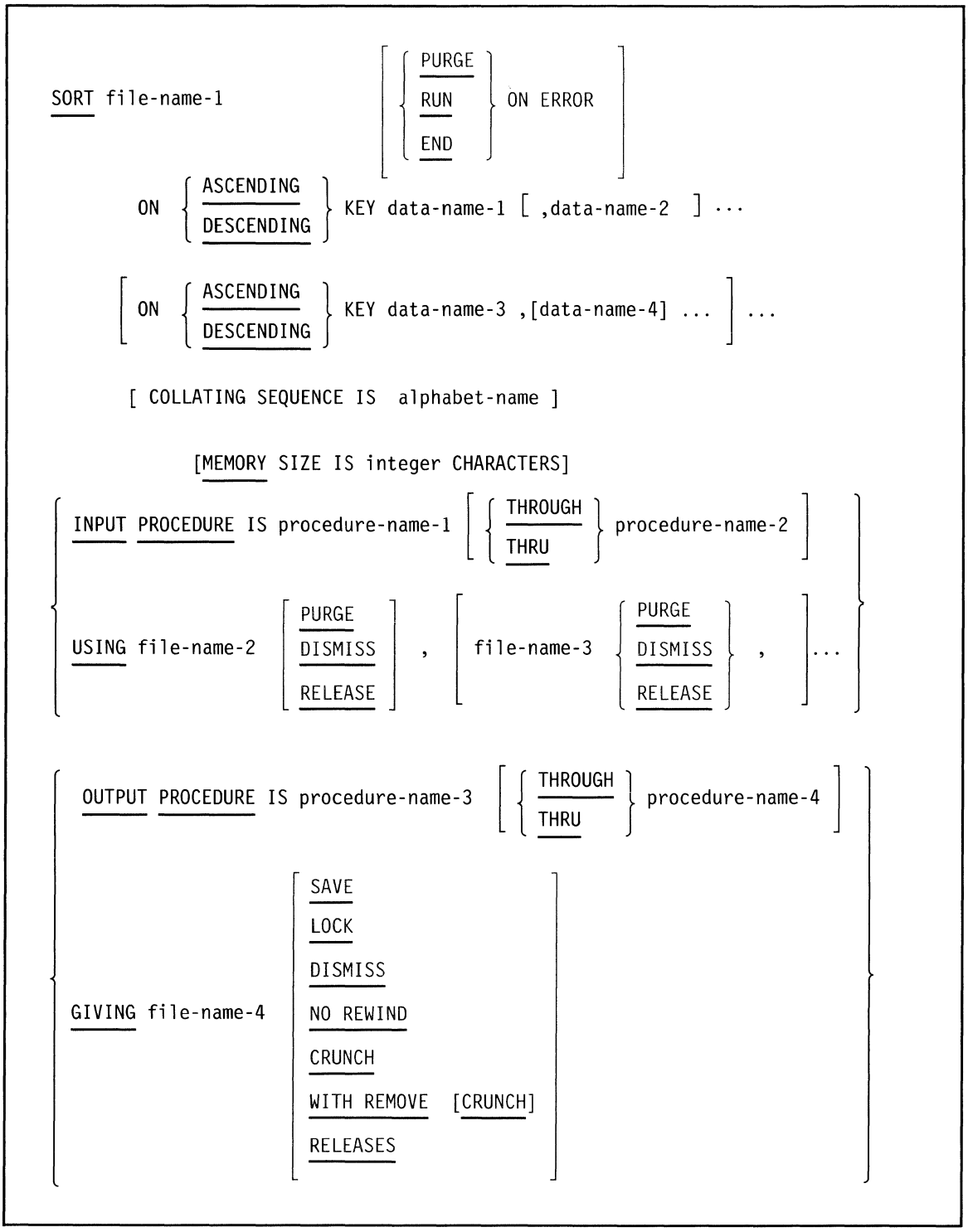
In Format 2, the contents of index-name-4 are incremented (UP BY) or decremented (DOWN BY) by a value that corresponds to the number of occurrences represented by the value of integer-2 or identifier-4. Thereafter, the process is repeated for index-name-5, and so on. The value of identifier-4 remains unchanged.

SORT Statement

The SORT statement sorts records by executing input procedures or by transferring records from another file. It sorts the records in the sort file on a set of specified keys, and in the final phase of the sort operation, makes each record from the sort file available, in sorted order, to some output procedures or to an output file.

Note: *A SORT statement cannot appear in the Declaratives section of a program.*

The following is the general format:


SORT file-name-1
PURGE
RUN
END
ON ERROR
ON ASCENDING DESCENDING KEY data-name-1 [,data-name-2] ...
ON ASCENDING DESCENDING KEY data-name-3 ,[data-name-4] ... ...
[COLLATING SEQUENCE IS alphabet-name]
[MEMORY SIZE IS integer CHARACTERS]
INPUT PROCEDURE IS procedure-name-1 THROUGH THRU procedure-name-2
USING file-name-2 PURGE DISMISS RELEASE , file-name-3 PURGE DISMISS RELEASE , ...
OUTPUT PROCEDURE IS procedure-name-3 THROUGH THRU procedure-name-4
GIVING file-name-4 SAVE LOCK DISMISS NO REWIND CRUNCH WITH REMOVE [CRUNCH] RELEASES

SORT statements can appear anywhere except in the declarative portion of the Procedure Division or in an input or output procedure associated with a SORT or MERGE statement.

When the SORT statement is executed, the MCP intrinsic *SORT:* is invoked. The total block size of the SD file must not exceed 19998 bytes.

Input to and from the sort can optionally be variable-length tape records. If input is variable-length, output must be variable-length also. The record length occupies the first 4 bytes of each record, in 8-bit format (PIC 9(4) USAGE DISPLAY).

You must ensure that the length of the sort file is sufficient (in the SD, the value of AREAS times the value of AREASIZE) to hold the file to be sorted. If the sort file size is too small, the program can fail and the MCP message EOF NO LABEL appears when the sort work file overflows. Specifying a sort file size of greater than 10 million records results in a run-time warning requiring manual intervention.

No more than one file-name from any one multiple file reel can appear in the SORT statement.

File-name-1 must be described in a sort-merge file description entry in the Data Division.

File-name-2, file-name-3, and file-name-4 must be described in the Data Division in a file description entry, not in a sort-merge file description entry. The actual size of the logical record or records described for file-name-2, file-name-3, and file-name-4 must be equal to the actual size of the logical record or records described for file-name-1. If the data descriptions of the elementary items that make up these records are not identical, you must describe the corresponding records so they are allowed an equal number of character positions.

The maximum number of records in a file to be sorted is 10 million. The maximum number of KEY items that can be specified is 40. The maximum length of each key is 98 bytes (98 digits if COMPUTATIONAL). The maximum total length of the sort key (the sum of the lengths of all KEY items in a SORT statement) is 290 bytes; however, sort requires more processor time when the total key length exceeds 99 bytes.

The ON ERROR option provides programmers some control over irrecoverable parity errors when INPUT/OUTPUT PROCEDURES are not present in a program. The following are the purposes of the options that can be used with the ON ERROR clause.

- PURGE drops all records in a block containing an irrecoverable parity error. Processing continues after the ODT displays a message giving the relative position in the file of the bad block.
- RUN allows the program to use the bad block and provides the same message as defined for PURGE.
- END terminates the program. It is assumed if no option is specified.

Data-name-1, data-name-2, data-name-3, and data-name-4 are KEY data-names. They are subject to the following rules:

- The data items identified by KEY data-names must be described in records associated with file-name-1.
- KEY data-names can be qualified.
- The data items identified by KEY data-names must not be variable length.
- The data items identified by KEY data-names when file-name-1 has more than one record description can all be described within one of the record descriptions or in any combination of record descriptions. Redescribing the KEY data-names in each record description is not necessary.
- None of the data items identified by KEY data-names can be described by an entry that either contains an OCCURS clause or is subordinate to an entry that contains one.

The data-names following the word KEY are listed in the SORT statement from left to right and in order of decreasing significance no matter how they are divided into KEY phrases. In the format, data-name-1 is the major key, and data-name-2 is the next most significant key, followed by data-name-3 and data-name-4.

- When the ASCENDING phrase is specified, the sorted sequence is from the lowest to the highest value of the contents of the data items identified by the KEY data-names, according to the rules for comparing operands in a relation condition.
- When the DESCENDING phrase is specified, the sorted sequence is from the highest to the lowest value of the contents of the data items identified by the KEY data-names, according to the rules for comparing operands in a relation condition.

If the COLLATING SEQUENCE clause is used, alphabet-name must be specified in the SPECIAL-NAMES paragraph of the Environment Division.

MEMORY SIZE is used as a guideline allocating SORT memory area. If record size for the SORT file is greater than or equal to 4000 characters, memory size must be greater than or equal to 100,000 characters.

Procedure-name-1 represents the name of an input procedure and procedure-name-3 represents the name of an output procedure.

If an INPUT PROCEDURE is specified, control passes to the input procedure before file-name-1 is sequenced by the SORT statement. The compiler inserts a return mechanism at the end of the last paragraph or section in the input procedure. When control passes the last statement in the input procedure, the records that have been released to file-name-1 are sorted.

The input procedure must consist of one or more paragraphs or sections that appear contiguously in a source program and do not form a part of any output procedure. To transfer records to the file referenced by file-name-1, you must include in the input procedure the execution of at least one RELEASE statement. Do not pass control to the input procedure except when a related SORT statement is being executed. The input procedure can include any procedures needed to select, create, or modify records. The restrictions on the procedural statements within the input procedure are as follows:

- The input procedure must not contain any SORT or MERGE statements.
- The input procedure must not contain any explicit transfers of control to points outside the input procedure; ALTER, GO TO, and PERFORM statements in the input procedure are not permitted to refer to procedure-names outside the input procedure. COBOL statements are allowed that cause an implied transfer of control to declaratives.
- The remainder of the Procedure Division must not contain any transfers of control to points inside the input procedure; ALTER, GO TO and PERFORM statements in the remainder of the Procedure Division must not refer to procedure-names within the input procedure.

Up to eight files can be specified in a USING phrase. If the USING phrase is specified, all the records in file-name-2 and file-name-3 are transferred automatically to file-name-1. At the time the SORT statement is executed, file-name-2 and file-name-3 must not be open. The SORT statement automatically initiates the processing of, makes available the logical records for, and terminates the processing of file-name-2 and file-name-3. These implicit functions are performed so that any associated USE procedures are executed.

The terminating function for all files is performed as if a CLOSE statement, without optional phrases, had been executed for each file. The SORT statement also automatically performs the implicit functions of moving the records from the file area of file-name-2 and file-name-3 to the file area for file-name-1 and obtains the release of records to the initial phase of the sort operation.

If an output procedure is specified, control passes to procedure-name-3 after the SORT statement sequences file-name-1. The compiler inserts a return mechanism at the end of the last paragraph or section in the output procedure. When control passes the last statement in the output procedure, the return mechanism terminates the sort and then passes control to the next executable statement after the SORT statement. Before entering the output procedure, the sort procedure reaches a point at which it can select the next record in sorted order when requested. The RETURN statements in the output procedure are the requests for the next record.

The output procedure must consist of one or more paragraphs or sections that appear contiguously in a source program and do not form part of any input procedure. To make sorted records available for processing, you must include in the output procedure the execution of at least one RETURN statement. Control must not be passed to the output procedure except when a related SORT statement is being executed. The output procedure can consist of any procedures needed to select, modify, or copy the records that are being returned, one at a time in sorted order, from the sort file. The restrictions on the procedural statements within the output procedure are as follows:

- The output procedure must not contain any SORT or MERGE statements.
- The output procedure must not contain any explicit transfers of control to points outside the output procedure; ALTER, GO TO, and PERFORM statements in the output procedure cannot refer to procedure-names outside the output procedure. COBOL statements are allowed that cause an implied transfer of control to declaratives.
- The remainder of the Procedure Division must not contain any transfers of control to points inside the output procedure; ALTER, GO TO, and PERFORM statements in remainder of the Prodedure Division cannot refer to procedure-names within the output procedure.

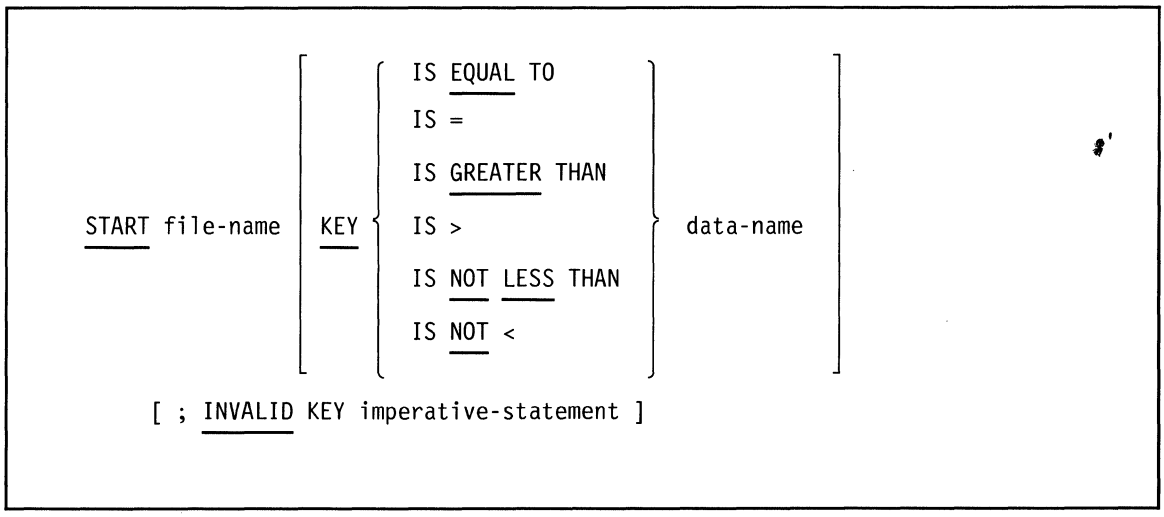
If the GIVING phrase is specified, all the sorted records in file-name-1 are automatically written to file-name-4 as the implied output procedure for this SORT statement. When the SORT statement is executed, file-name-4 must not be open. The SORT statement automatically initiates the processing of, releases the logical records to, and then terminates the processing of file-name-4. These implicit functions are performed so that any associated USE procedures are executed. The terminating function is performed as if a CLOSE statement, without optional phrases, had been executed for the file. The SORT statement also automatically performs the implicit functions of the return of the sorted records from the final phase of the sort operation and moves the records from the file area for file-name-1 to the file area for file-name-4.

Use the PURGE, DISMISS, or RELEASE options to specify the type of file close for the USING files, file-name-2, and file-name-3. The SAVE, LOCK, DISMISS, NO REWIND, CRUNCH, WITH REMOVE, WITH REMOVE CRUNCH, and RELEASE options can be used to specify the type of file close for the GIVING file, file-name-4 (see "CLOSE Statement" in this section).

START Statement

The START statement provides a basis for logical positioning within a relative or indexed file, for subsequent sequential retrieval of records.

The following is the general format:



Note: *The required relational characters ">", "<" and "=" are not underlined to avoid confusion with other symbols such as "≥" (greater than or equal to).*

File-name must be the name of a RELATIVE or INDEXED file with sequential or dynamic access.

Data-name can be qualified.

The INVALID KEY phrase must be included if no applicable USE procedure is specified for file-name.

For a relative file, data-name, if specified, must be the data item used in the RELATIVE KEY phrase of the associated file control entry.

For an indexed file with the KEY phrase specified, data-name can refer to a data item specified as a record key associated with file-name. It can also refer to any data item of category alphanumeric subordinate to the data-name of a data item specified as a record key associated with file-name whose leftmost character position corresponds to the leftmost character position of that record key data item.

File-name must be open in the input or I/O mode when you execute the START statement.

If the KEY phrase is not specified, the relational operator IS EQUAL TO is implied.

The comparison specified by the relational operator in the KEY phrase includes the following:

- For a relative file, the data item referenced by the RELATIVE KEY clause associated with file-name
- For an indexed file with the KEY phrase specified, the data item referenced by data-name
- For an indexed file with the KEY phrase not specified, the data item referenced in the RECORD KEY clause associated with file-name

The type of comparison specified by the relational operator in the KEY phrase occurs between a key associated with a record in the file referenced by file-name and a data item as specified in the preceding paragraphs.

- The current record pointer is positioned to the first logical record that currently exists in the file whose key satisfies the comparison.
- If no record satisfies the comparison, an INVALID KEY condition exists, the execution of the START statement is unsuccessful, and the position of the current record pointer is undefined.

Executing the START statement updates the value of the FILE STATUS data item associated with file-name.

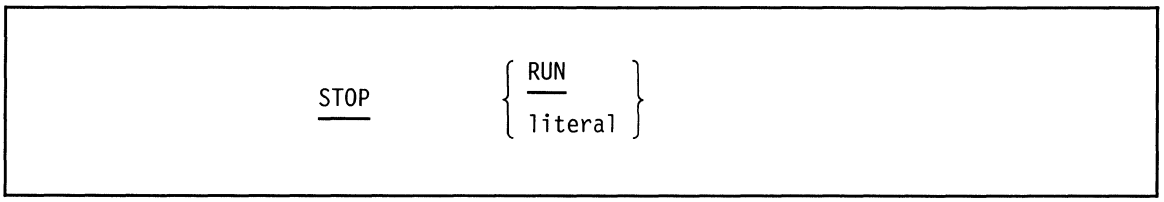
The following applies to indexed files:

- If the execution of the START statement is successful, a key of reference is established and used in subsequent Format 1 READ statements, as follows:
  - If the KEY phrase is not specified, the prime record key for file-name becomes the key of reference.
  - If the KEY phrase is specified and data-name is used as a record key for file-name, that record key becomes the key of reference.
  - If the KEY phrase is specified and data-name is not used as a record key for file-name, the record key whose leftmost character position corresponds to the leftmost character position of the data item specified by data-name becomes the key of reference.
- If the execution of the START statement is not successful, the key of reference is undefined.

STOP Statement

The STOP statement permanently or temporarily suspends the execution of the object program.

The following is the general format:



If a STOP RUN statement appears in a consecutive sequence of imperative statements within a sentence, it must appear as the last statement.

The following is true if the RUN phrase is used:

1. IX1 is set to zero.
2. The MCP issues a simple CLOSE to all files left open.
3. The program terminates.

The following is true if the STOP literal is specified:

1. The ODT displays the literal.
2. The MCP displays ** *program-id* = *mix ACCEPT*
3. The program pauses its execution.
4. The program resumes when the operator enters *mix AX*

The literal can be numeric, nonnumeric, or any figurative constant except ALL. If the literal is numeric, then it must be an unsigned integer.

Continuation of the object program begins with the execution of the next executable statement in sequence.

Clearing of IX1 at STOP RUN (COMPILE)

Code is generated to clear IX1 to zeros when the STOP RUN statement is executed. Doing so is meaningful only when a user program is initiated by a COMPILE command rather than an EXECUTE command.

The program operates in the same way as it would if it were initiated by an EXECUTE command. At EOJ, however, the MCP inspects the IX1 of the program. If IX1 is non-zero when a COMPILE-initiated program terminates, the MCP displays the message ** *SYNTAX ERROR* and enters a run log entry showing that the program terminated with syntax errors. (Thus, compilers always clear IX1 when there are no syntax errors and enter a value into IX1 when there are syntax errors.)

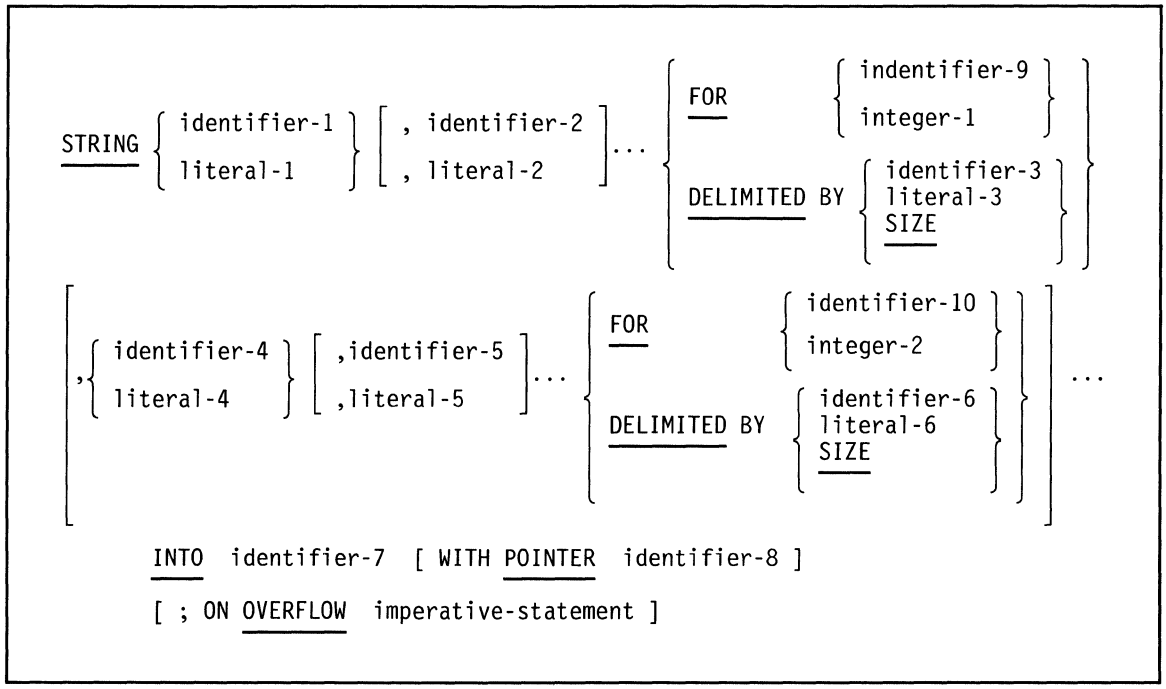
The fact that STOP RUN causes IX1 to be cleared to zeros means that you can initiate programs with the COMPILE command and be assured that they will terminate with a normal end-of-job status. (There is no other COBOL syntax to clear IX1.)

If a program must terminate with a syntax error condition, the program must not execute a STOP RUN. An alternative, such as CALL "INTRINSIC ABORT", must be used so that IX1 is not cleared to zeros; and the program must ensure that IX1 is non-zero. Certain operations, such as table searching and indexing, can cause values to be placed into IX1.

STRING Statement

The STRING statement provides juxtaposition of the partial or complete contents of two or more data items into a single data item.

The following is the general format:


STRING { identifier-1 / literal-1 } [, identifier-2 / , literal-2] ...
{ FOR { indentifier-9 / integer-1 } / DELIMITED BY { identifier-3 / literal-3 / SIZE } }
[, { identifier-4 / literal-4 } [,identifier-5 / ,literal-5] ...
{ FOR { identifier-10 / integer-2 } / DELIMITED BY { identifier-6 / literal-6 / SIZE } }] ...
INTO identifier-7 [WITH POINTER identifier-8]
[; ON OVERFLOW imperative-statement]

Each literal can be any figurative constant without the optional word ALL.

All literals must be described as nonnumeric literals, and all identifiers except identifier-8, identifier-9, and identifier-10 must be described implicitly or explicitly as USAGE IS DISPLAY.

Where identifier-1, identifier-2, or identifier-3 is an elementary numeric data item, it must be described as an integer without the symbol *P* in its PICTURE character-string.

All references to identifier-1, identifier-2, identifier-3, identifier-9, literal-1, literal-2, literal-3, and integer-1 apply equally to identifier-4, identifier-5, identifier-6, identifier-10, literal-4, literal-5, literal-6, and integer-2, respectively, and all recursions thereof.

Identifier-1, literal-1, identifier-2, and literal-2 represent the sending items. Identifier-7 represents the receiving item.

Literal-3 or identifier-3 indicates the character or characters delimiting the move. If the SIZE phrase is used, the complete data item defined by identifier-1, literal-1, identifier-2, and/or literal-2 is moved. When you use a figurative constant as a delimiter, it stands for a 1-character nonnumeric literal. When a figurative constant is specified as literal-1, literal-2, or literal-3, the constant refers to an implicit 1-character data item whose usage is DISPLAY.

Identifier-7 must represent an elementary alphanumeric data item without editing symbols or the JUSTIFIED clause.

Identifier-8 must represent an elementary numeric integer data item of sufficient size to contain a value equal to the size plus 1 of the area referenced by identifier-7. The symbol *P* cannot be used in the PICTURE character-string of identifier-8.

Integer-1 or identifier-9 indicates the number of characters to be moved.

Identifier-9 and identifier-10 must represent elementary numeric integer data items.

When the STRING statement is executed, the following rules govern the transfer of data:

- Those characters from literal-1 or literal-2 or from the contents of the data item referenced by identifier-1 or identifier-2 are transferred to the contents of identifier-7. However, no space-filling is provided (see "MOVE Statement," earlier in this section.)
- If the DELIMITED phrase is specified without the SIZE phrase, the contents of the data item referenced by identifier-1 and identifier-2 or the value of literal-1 and literal-2 are transferred to the receiving data item. The transfer occurs in the sequence specified in the STRING statement, beginning with the leftmost character and continuing from left to right until the end of the data item or until the character or characters specified by literal-3 or by the contents of identifier-3 are encountered. The character or characters specified by literal-3 or by the data item referenced by identifier-3 are not transferred.

  Note: *If identifier-1 or identifier-2 begins with the delimiting character or characters, no data is transferred from these items.*

- If the DELIMITED phrase is specified with the SIZE phrase, the entire contents of literal-1, literal-2 or the contents of the data item referenced by identifier-1, identifier-2 are transferred. This transfer occurs in the sequence specified in the STRING statement, to the data item referenced by identifier-7 until all data is transferred or until the end of the data item referenced by identifier-7 is reached.

- If the FOR phrase is specified, the contents of the data item referenced by identifier-1 and identifier-2 or the value of literal-1 and literal-2 are transferred to the receiving data item in the sequence specified in the STRING statement. The transfer begins with the leftmost character and continues from left to right until the end of the data item is reached, until the number of characters specified by integer-1 or by the contents of identifier-9 are moved, or until the end of identifier-7.

  If the length of the sending data item is less than the length specified in the FOR phrase, the sending item will be padded with spaces.

If you specify the POINTER phrase, identifier-8 is explicitly available so you can set its initial value. The initial value must not be less than one.

If you do not use the POINTER phrase, the following general rules apply as if you had specified identifier-8 with an initial value of 1.

- When characters are transferred to the data item referenced by identifier-7, the moves behave as though the characters were moved one at a time from the source into the character position of the data item referenced by identifier-7 designated by the value associated with identifier-8, and then identifier-8 increased by one prior to the move of the next character. The value associated with identifier-8 is changed during execution of the STRING statement only by the behavior previously specified.
- When the STRING statement has been executed, only the portion of the data item referenced by identifier-7 is changed. All other portions of the data item referenced by identifier-7 contain data that was present before the execution.
- If after initialization of the STRING statement and before execution of the STRING statement is completed, the value associated with identifier-8 is either less than one or exceeds the number of character positions in the data item referenced by identifier-7, then:
  - No further data is transferred to the data item referenced by identifier-7.
  - The imperative statement in the ON OVERFLOW phrase is executed, if specified.
  - If the ON OVERFLOW phrase is not used, control is transferred to the next executable statement.
  - If the ON OVERFLOW phrase is used, the POINTER phrase must also be used.

Refer to example 1 in Appendix E for use of the STRING statement.

SUBTRACT Statement

The SUBTRACT statement subtracts either one or the sum of two or more numeric data items from one or more items and then sets the values of one or more items equal to the results.

The following is the general format:

Format 1

```
SUBTRACT { identifier-1 } [ , identifier-2 ] ... FROM identifier-m [ROUNDED]
         { literal-1    }  [ , literal-2    ]

[, identifier-n [ROUNDED] ] ... [; ON SIZE ERROR imperative-statement]
```

Format 2

```
SUBTRACT { identifier-1 } [ , identifier-2 ] ... FROM { identifier-m }
         { literal-1    }  [ , literal-2    ]        { literal-m    }

    GIVING identifier-n [ROUNDED] [, identifier-o [ROUNDED] ] ...

    [; ON SIZE ERROR imperative-statement]
```

Format 3

```
SUBTRACT { CORRESPONDING } identifier-1 FROM identifier-2 [ROUNDED]
         { CORR          }

    [; ON SIZE ERROR imperative-statement]
```

In Format 1, all literals or identifiers preceding the word FROM are added together. This total is subtracted from the current value of identifier-m, and the results are stored immediately into identifier-m. This process is repeated for each operand following the word FROM.

In Format 2, all literals or identifiers preceding the word FROM are added together, the sum is subtracted from literal-m or identifier-m, and the result of the subtraction is stored as the new value of identifier-n and identifier-o.

If Format 3 is used, data items in identifier-1 are subtracted from and stored into corresponding data items in identifier-2.

Each identifier must refer to a numeric elementary item with the following exceptions:

- In Format 2, each identifier following the word GIVING must refer to either an elementary numeric or numeric-edited item.
- In Format 3, each identifier must refer to a group item.

Each literal must be a numeric literal.

Additional rules and explanations relative to this statement are given in the appropriate paragraphs. See "Intermediate Data Item", "ROUNDED Phrase", "SIZE ERROR Phrase", "CORRESPONDING Phrase", "Arithmetic Statements", "Overlapping Operands", and "Multiple Results in Arithmetic Statements" in this section.

UNSTRING Statement

The UNSTRING statement separates contiguous data in a sending field and places it into multiple receiving fields.

The following are the general formats:

Format 1

```
UNSTRING identifier-1
[ DELIMITED BY [ALL] { identifier-2 } [ , OR [ALL] { identifier-3 } ] ... ]
                     { literal-1   }               { literal-2   }
INTO identifier-4 [,DELIMITER IN identifier-5] [,COUNT IN identifier-6]

[ ,identifier-7 [,DELIMITER IN identifier-8]  [ , COUNT IN identifier-9] ] ...
[WITH POINTER identifier-10] [TALLYING IN identifier-11]
[; ON OVERFLOW imperative-statement]
```

Format 2

```
UNSTRING identifier-1
  INTO identifier-4 FOR { identifier-12 }
                        { literal-3     }
[ , identifier-7 FOR { identifier-13 } ]    ...
                     { literal-4     }
[WITH POINTER identifier-10] [TALLYING IN identifier-11]
[; ON OVERFLOW imperative-statement]
```

Each literal must be a nonnumeric literal. In addition, each literal can be any figurative constant without the optional word ALL.

No identifier can name a level 88 entry.

All references to identifier-2, literal-1, identifier-4, identifier-5, identifier-6, identifier-12, and literal-3 apply equally to identifier-3, literal-2, identifier-7, identifier-8, identifier-9, identifier-13, and literal-4 respectively. Describe the identifiers as follows:

- Identifier-1, identifier-2, identifier-3, identifier-5, and identifier-8 must each be, implicitly or explicitly, an alphanumeric data item.
  - Identifier-4 and identifier-7 can be either alphabetic (except that the symbol *B* cannot be used in the PICTURE character-string), alphanumeric, or numeric (except that the symbol *P* cannot be used in the PICTURE character-string) and must be USAGE IS DISPLAY.
- Identifier-6, identifier-9, identifier-10, and identifier-11 must be elementary numeric integer data items (except that the symbol *P* cannot be used in the PICTURE character-string).
- Identifier-12 and identifier-13 must be elementary numeric integer data items (except that the symbol *P* cannot be used in the PICTURE character-string).

The evaluation of subscripting and indexing for the identifiers is as follows:

- Any subscripting or indexing associated with identifier-1, identifier-10, or identifier-11 is evaluated only once, immediately before any data is transferred by executing the UNSTRING statement.
- Any subscripting or indexing associated with identifier-2, identifier-3, identifier-4, identifier-5, or identifier-6 is evaluated immediately before the transfer of data into the respective data item.

Identifier-1 represents the sending area.

Literal-1 or the contents of the data item referenced by identifier-2 can contain any character in the computer's character set.

Each literal-1 or data item referenced by identifier-2 represents one delimiter. When a delimiter contains two or more characters, all of the characters must be present, in the order given, in contiguous positions of the sending item, to be recognized as a delimiter.

When a figurative constant is used as a delimiter, it stands for a 1-character nonnumeric literal.

When the ALL phrase is specified, one occurrence or two or more contiguous occurrences of literal-1 (figurative constant or not) or the contents of the data item referenced by identifier-2 are treated as if it were only one occurrence, and this occurrence is moved to the receiving data item according to the fourth item under "Rules for Data Transfer" later in this section.

Without the ALL phrase, when any examination encounters two contiguous delimiters, the current receiving area is either space or zero filled according to the description of the receiving area.

When two or more delimiters are specified in the DELIMITED BY phrase, an OR condition exists between them. Each delimiter is compared to the sending field. If a match occurs, the character or characters in the sending field is considered to be a single delimiter. No character in the sending field can be considered a part of more than one delimiter.

Each delimiter is applied to the sending field in the sequence specified in the UNSTRING statement.

You can use the DELIMITER IN phrase and the COUNT IN phrase only if the DELIMITED BY phrase is specified.

Identifier-4 represents the data receiving area. Identifier-5 represents the receiving area for delimiters.

Literal-1 or the data item referenced by identifier-2 specifies a delimiter.

Identifier-6 represents the count of the number of characters within the data item referenced by identifier-1 isolated by the delimiters for the move to identifier-4. This value does not include a count of the delimiter character or characters.

The data item referenced by identifier-10 contains a value that indicates a relative character position within the area defined by identifier-1.

The contents of the data item referenced by identifier-10 is incremented by one for each character examined in the data item referenced by identifier-1. When the execution of an UNSTRING statement with a POINTER phrase is completed, the contents of the data item referenced by identifier-10 contains a value equal to the initial value plus the number of characters examined in the data item referenced by identifier-1.

The data item referenced by identifier-11 is a counter that records the number of data items acted upon during the execution of an UNSTRING statement.

When the execution of an UNSTRING statement with a TALLYING phrase is completed, the contents of the data item referenced by identifier-11 contains a value equal to its initial value plus the number of data receiving items acted upon.

It is the user's responsibility to initialize the contents of the data items associated with the POINTER phrase or the TALLYING phrase.

Literal-3 or the data item referenced by identifier-12 specifies a count of characters within identifier-1 that is moved to identifier-4. If the number of characters remaining in the data item referenced by identifier-1 is less than the number of characters specified by literal-3 or the data item referenced by identifier-12, then the short field is transferred according to item 3 under "Rules for Data Transfer" later in this section.

Any of the following situations causes an overflow condition:

- An UNSTRING is initiated. The value in the data item referenced by identifier-10 is less than 1 or greater than the size of the data item referenced by identifier-1.
- During execution of an UNSTRING statement, all data receiving areas have been acted upon. The data item referenced by identifier-1 contains characters that have not been examined.
- An UNSTRING is initiated. The value in the data item referenced by identifier-12 is less than 1 or greater than the size of the data item referenced by identifier-1.
- During execution of an UNSTRING statement, all data receiving areas have been acted upon. The number of characters acted upon is less than the value of identifier-12 or literal-3.

When an overflow condition exists, the UNSTRING operation is terminated. If an ON OVERFLOW phrase was specified, the imperative statement included in the ON OVERFLOW phrase is executed. If the ON OVERFLOW phrase was not specified, control is transferred to the next executable statement.

Rules for Data Transfer

When the UNSTRING statement is initiated, the current receiving area is the data item referenced by identifier-4. Data is transferred from the data item referenced by identifier-1 to the data item referenced by identifier-4 according to the following rules:

- If the POINTER phrase is specified, the string of characters referenced by identifier-1 is examined beginning with the relative character position indicated by the contents of the data item referenced by identifier-10. If the POINTER phrase is not specified, the string of characters is examined beginning with the leftmost character position.
- If the DELIMITED BY phrase is specified, the examination proceeds left to right until either a delimiter specified by the value of literal-1 or the data item referenced by identifier-2 is encountered. If the DELIMITED BY phrase is not specified, the number of characters examined is equal to the size of the current receiving area. However, if the sign of the receiving item is defined as occupying a separate character position, the number of characters examined is one less than the size of the current receiving area.

  If the end of the data item referenced by identifier-1 is encountered before the delimiting condition is met, the examination terminates with the last character examined.
- The characters thus examined (excluding any delimiting characters) are treated as an elementary alphanumeric data item and are moved into the current receiving area, according to the rules for the MOVE statement.

Note: *If delimiters have been specified and identifier-1 begins with the specified delimiter or delimiters, the first receiving field, identifier-4, is either zero or space filled, according to the description of identifier-4. If this value is not desired, the user can first INSPECT identifier-1 tallying the leading delimiter or delimiters, and then unstring identifier-1, using the POINTER clause wherein identifier-10 was set to 1 more than the count tallied by the INSPECT clause.*

- If the DELIMITER IN phrase is specified, the delimiting character or characters are treated as an elementary alphanumeric data item and are moved into the data item referenced by identifier-5, according to the rules for the MOVE statement. If the delimiting condition is the end of the data item referenced by identifier-1, then the data item referenced by identifier-5 is space-filled.
- If the COUNT IN phrase is specified, a value equal to the number of characters thus examined (excluding any delimiter character) is moved into the area referenced by identifier-6, according to the rules for an elementary move.
- If the DELIMITED BY phrase is specified, the string of characters is further examined beginning with the first character to the right of the delimiter. If the DELIMITED BY phrase is not specified, the string of characters is further examined beginning with the character to the right of the last character transferred.
- After data is transferred to the data item referenced by identifier-4, the current receiving area is the data item referenced by identifier-7. The actions described in the preceding are repeated until either all the characters are exhausted in the data item referenced by identifier-1 or until there are no more receiving areas.

Refer to example 1 in Appendix E for an illustration of the use of the UNSTRING statement.

USE Statement

The USE statement Format 1 identifies the user items to be monitored by the associated debugging section.

The USE statement Format 2 specifies for input/output error handling procedures that are in addition to the standard procedures provided by the input/output control system.

The USE statement Format 3 specifies the procedure for handling program errors.

The USE statement Format 4 specifies the procedures for handling tape labels.

The following are the general formats:

Format 1

```
USE FOR DEBUGGING ON { [ ALL REFERENCES OF] identifier-1 }
                     { file-name-1                       }
                     { procedure-name-1                  }
                     { ALL PROCEDURES                    }

    [ , [ALL REFERENCES OF ] identifier-2 ]
    [ , file-name-2                       ]
    [ , procedure-name-2                  ]  ... .
    [ , ALL PROCEDURES                    ]
```

Format 2

```
USE AFTER STANDARD { EXCEPTION } PROCEDURE ON { file-name-1                  }
                   { ERROR     }              {       [,file-name-2]...    }
                                              { INPUT                      }
                                              { OUTPUT                     } .
                                              { I-O                        }
                                              { EXTEND                     }
```

Format 3

```
USE ON PROGRAM-FAULT.
```

Format 4

```
USE { AFTER  } STANDARD { BEGINNING } [ { FILE } ]
    { BEFORE }          { ENDING    } [ { REEL } ]

                  { INPUT                                  }
LABEL PROCEDURE ON { OUTPUT                                 } .
                  { file-name-1 [ , file-name-2 ] ...      }
```

Format 1

Debugging sections, if specified, must appear together immediately after the DECLARATIVES header. Except in the USE FOR DEBUGGING statement itself, there must be no reference to any nondeclarative procedure within the debugging section.

Statements appearing outside the set of debugging sections must not reference procedure-names defined within the set of debugging sections.

Any given identifier, file-name, or procedure-name can appear in only one USE FOR DEBUGGING statement and can appear only once in that statement.

In the following general rules, all references to identifier-1, file-name-1, and procedure-name-1, apply equally to identifier-2, file-name-2, and procedure-name-2. A reference to identifier-1, file-name-1, or procedure-name-1 does not constitute a reference to that item for the debugging described in the following paragraphs.

The automatic execution of a debugging section is not caused by a statement appearing in that section.

The associated debugging section is not executed for a specific operand more than once as a result of executing a single statement, regardless of the number of times that operand is explicitly specified. In the case of a PERFORM statement that

causes iterative execution of a referenced procedure, the associated debugging section is executed once for each iteration.

Within an imperative statement, each individual occurrence of an imperative verb identifies a separate statement for the purpose of debugging.

Monitoring Data Items

When the ALL REFERENCES OF identifier-1 phrase is specified, that debugging section is executed for every statement that explicitly references identifier-1 at each of the following times:

- In the case of a WRITE or REWRITE statement, immediately before executing that WRITE or REWRITE statement and after executing any implicit move resulting from the presence of the FROM phrase
- In the case of a GO TO statement with a DEPENDING ON phrase, immediately before control is transferred and prior to executing any debugging section associated with the procedure-name to which control is to be transferred
- In the case of a PERFORM statement in which a VARYING, AFTER, or UNTIL phrase refers to identifier-1, immediately after each initialization, modification, or evaluation of the contents of the data item referenced by identifier-1
- In the case of any other COBOL statement, immediately after execution of that statement

Note: *If identifier-1 is specified in a phrase that is not executed or evaluated, the associated debugging section is not executed.*

When identifier-1 is specified without the ALL REFERENCES OF phrase, that debugging section is executed at each of the following times:

- In the case of a WRITE or REWRITE statement that explicitly refers to identifier-1, immediately before executing that statement and after executing any implicit move resulting from the presence of the FROM phrase
- In the case of a PERFORM statement in which a VARYING, AFTER or UNTIL phrase refers to identifier-1, immediately after each initialization, modification, or evaluation of the contents of the data item referenced by identifier-1
- Immediately after executing any other COBOL statement that explicitly refers to and changes the contents of the data item referenced by identifier-1

Note: *If the identifier is specified in a phrase that is not executed or evaluated, the associated debugging section is not executed.*

If the data description entry of the data item referenced by identifier-1 contains an OCCURS clause or is subordinate to a data description entry that contains an OCCURS clause, you must specify identifier-1 without the subscripting or indexing normally required.

Monitoring Files

When file-name-1 is specified in a USE FOR DEBUGGING statement, that debugging section is executed at the following times:

- After executing any OPEN or CLOSE statement that refers to file-name-1
- After executing any READ statement (after any other specified USE procedure) not resulting in execution of an associated AT END or INVALID KEY imperative statement
- After executing any DELETE, SEEK, or START statement that refers to file-name-1

Monitoring Procedures

When procedure-name-1 is specified in a USE FOR DEBUGGING statement, that debugging section is executed at the following times:

- Immediately before each execution of the named procedure
- Immediately after executing an ALTER statement that refers to procedure-name-1

The ALL PROCEDURES phrase can appear only once in a program. When you specify the ALL PROCEDURES phrase, you must not include procedure-name-1 and procedure-name-2 in any USE FOR DEBUGGING statement.

DEBUG-ITEM

Associated with each execution of a debugging section is the special register DEBUG-ITEM, which provides information about the conditions that caused the execution of a debugging section. DEBUG-ITEM has the following implicit description:

```
01  DEBUG-ITEM.
    02  DEBUG-LINE     PIC X(6).
    02  FILLER         PIC X VALUE SPACE
    02  DEBUG-NAME     PIC X(30).
    02  FILLER         PIC X  VALUE SPACE.
    02  DEBUG-SUB-1    PIC S9(4)  SIGN IS LEADING SEPARATE CHARACTER
    02  FILLER         PIC X  VALUE SPACE.
    02  DEBUG-SUB-2    PIC S9(4) SIGN IS LEADING SEPARATE CHARACTER
    02  FILLER         PIC X  VALUE SPACE.
    02  DEBUG-SUB-3    PIC S9(4)  SIGN IS LEADING SEPARATE CHARACTER
    02  FILLER         PIC X  VALUE SPACE.
    02  DEBUG-CONTENTS PIC X(n).
```

References to the special register DEBUG-ITEM can only be made from within a debugging section.

Prior to each execution of a debugging section, the contents of the data item referenced by DEBUG-ITEM are space-filled. The contents of data items subordinate to DEBUG-ITEM are then updated according to the general rules described in these paragrapgs and the following list, immediately before control is passed to that debugging section. The contents of any data item not specified in the following general rules remain spaces.

Updating occurs according to the rules for the MOVE statement. The only exception is that the move to DEBUG-CONTENTS is treated exactly as if it were an alphanumeric-to-alphanumeric elementary move with no conversion of data from one form of internal representation to another.

DEBUG-LINE is a data item that identifies the particular source statement that executed the debugging section. It contains the line number of the source image, a number that starts at one and is incremented by one for every source image compiled. This line number is printed on the source listing during compilation.

DEBUG-NAME contains the first 30 characters of the name that caused the debugging section to be executed.

All qualifiers of the name are separated in DEBUG-NAME by the word IN or OF. Subscripts or indices, if any, are not entered into DEBUG-NAME.

If the reference to a data item that causes the debugging section to be executed is subscripted or indexed, the occurrence number of each level is entered in DEBUG-SUB-1, DEBUG-SUB-2, and DEBUG-SUB-3 respectively, as necessary. If the data item is subscripted with more than three subscripts or indices, only the occurrence numbers of the first three levels are entered into the DEBUG-ITEM.

DEBUG-CONTENTS is a data item that is large enough to contain the data required by the following general rules:

- If a reference to identifier-1 causes the debugging section to be executed, then the following is true:
  - DEBUG-LINE identifies the source statement that refers to identifier-1.
  - DEBUG-NAME contains the name of identifier-1.
  - DEBUG-CONTENTS contains the contents of the data item referenced by identifier-1 at the time control passes to the debugging section.
- If references to file-name-1 causes the debugging section to be executed, then the following is true:
  - DEBUG-LINE identifies the source statement that refers to file-name-1.
  - DEBUG-NAME contains the name of file-name-1.
  - For READ, DEBUG CONTENTS contains the entire record read.
  - For all other references to file-name-1, DEBUG-CONTENTS contains spaces.
- If procedure-name-1 is a USE procedure to be executed, the following conditions exist:

  - DEBUG-LINE identifies the statement that executes the USE procedure.
  - DEBUG-NAME contains procedure-name-1.
  - DEBUG-CONTENTS contains USE PROCEDURE.
- If a reference to procedure-name-1 in an ALTER statement causes the debugging section to be executed, the following conditions exist:
  - DEBUG-LINE identifies the ALTER statement that refers to procedure-name-1.
  - DEBUG-NAME contains procedure-name-1.
  - DEBUG-CONTENTS contains the applicable procedure-name associated with the TO phrase of the ALTER statement.
- If reference to procedure-name-1 in the INPUT or OUTPUT phrase of a SORT or MERGE statement causes the debugging section to be executed, the following conditions exist:
  - DEBUG-LINE identifies the SORT or MERGE statement that refers to procedure-name-1.
  - DEBUG-NAME contains procedure-name-1.
  - DEBUG-CONTENTS contains SORT INPUT if the reference to procedure-name-1 is in the INPUT phrase of a SORT statement.
  - DEBUG-CONTENTS contains SORT OUTPUT if the reference to procedure-name-1 is in the OUTPUT phrase of a SORT statement.
- If an implicit transfer of control from the previous sequential paragraph to procedure-name-1 causes the debugging section to be executed, the following conditions exist:
  - DEBUG-LINE identifies the previous statement.
  - DEBUG-NAME contains procedure-name-1.
  - DEBUG-CONTENTS contains FALL THROUGH.
- If the transfer of control from the control mechanism associated with a PERFORM statement caused the debugging section associated with procedure-name-1 to be executed, the following conditions exist:
  - DEBUG-LINE identifies the PERFORM statement that refers to procedure-name-1.
  - DEBUG-NAME contains procedure-name-1.
  - DEBUG-CONTENTS contains PERFORM LOOP.
- If the transfer of control associated with executing a GO TO statement causes the debugging section to be executed, the following conditions exist:
  - DEBUG-LINE identifies the GO TO statement whose execution transfers control to procedure-name-1.
  - DEBUG-NAME contains procedure-name-1.
- If the first execution of the first nondeclarative procedure in the program causes the debugging section to be executed, the following conditions exist:

- DEBUG-LINE identifies the first statement of that procedure.
- DEBUG-NAME contains the name of that procedure.
- DEBUG-CONTENTS contains START PROGRAM.

Format 2

This format specifies the procedures for I/O exception handling.

The words ERROR and EXCEPTION are synonymous. They can be used interchangeably.

The files implicitly or explicitly referenced in a USE statement need not all have the same organization or access.

The same file-name can appear in a different specific arrangement of the format. Appearance of a file-name in a USE statement must not cause the simultaneous request for execution of more than one USE procedure.

A USE statement, when present, must immediately follow a section header in the Declaratives section. It must also be followed by a period and then by a space. The remainder of the section must consist of zero or more procedural paragraphs that define the procedures to be used.

The USE statement itself is never executed. It merely defines the conditions calling for the execution of the USE procedure.

The MCP executes the USE procedures after completing the standard error retry routine or upon recognizing the AT END condition when the AT END phrase has not been specified in the input/output statement. The record in error has already been read. Thus another READ cannot appear in the USE procedure, since the MCP is already performing the section because of the previous READ.

After execution of a USE procedure, control returns to the invoking routine, that is, to the statement following the READ that detected the error condition. In the case of blocked or unblocked magnetic tape input, the tape is positioned ready to read the next record as soon as the USE procedure is completed. For example, if you wish to print the records that cannot be read because of an unrecoverable parity error, a procedure can be included, as illustrated by example 2 in Appendix E.

Within a USE procedure, no statement can be executed which would cause the execution of a USE procedure that had previously been invoked but had not yet returned control to the invoking routine.

Within a USE procedure, there must be no reference to any nondeclarative procedures. Conversely, in the nondeclarative portion there must be no reference to procedure-names that appear in the declarative portion, except that PERFORM statements can refer to a USE statement or to the procedures associated with such a USE statement.

Format 3

If USE ON PROGRAM-FAULT is specified for a procedure in the DECLARATIVES section, the program is reinstated in that procedure if it commits a fatal system error (such as an address error, invalid instruction, undigit arithmetic, and so forth).

A PROGRAM-FAULT procedure is subject to the same restrictions as other USE procedures, except that after execution of the PROGRAM-FAULT procedure, the program is terminated.

The USE statement, when present, must immediately follow a section header in the DECLARATIVES section. It must be followed by a period and a space, respectively. The remainder of the section consists of zero or more procedural paragraphs that define the procedures to be used. Only one USE ON PROGRAM-FAULT statement is permitted in a program.

Format 4

This format specifies procedures for tape file label handling.

The USE procedure is executed for beginning labels when you specify BEGINNING, and for ending labels when you specify ENDING.

If you do not specify FILE or REEL, the USE procedure is executed for both file and reel labels.

The files implicitly or explicitly referenced in a USE statement need not all have the same organization or access.

The same file-name can appear in a different specific arrangement of the format. Appearance of a file-name in a USE statement must not cause the simultaneous request for execution of more than one USE procedure.

Any file whose File Description entry specifies LABEL RECORDS ARE OMITTED is not included in INPUT or OUTPUT USE procedures.

The File Description entry for file-name-1 must not specify LABEL RECORDS ARE OMITTED.

The USE statement, when present, must immediately follow a section header in the DECLARATIVES section. It must be followed by a period and then by a space. The remainder of the section consists of zero or more procedural paragraphs that define the procedures to be used.

The USE statement itself is never executed. It merely defines the conditions calling for the execution of the USE procedures.

The USE AFTER STANDARD BEGINNING ... INPUT procedure is executed to access data on input tape beginning labels after they are read.

The USE AFTER STANDARD ENDING ... INPUT procedure is executed to access data on input tape ending labels after they are read.

The USE BEFORE STANDARD BEGINNING ... OUTPUT procedure is executed to access data on an output tape beginning label before it is written.

The USE BEFORE STANDARD ENDING ... OUTPUT procedure is executed to access data on an output tape ending label before it is written.

The USE AFTER STANDARD BEGINNING ... file-name-1 procedure is executed to access data on a tape beginning label after it is read or before it is written.

The USE BEFORE STANDARD ENDING ... file-name-1 procedure is executed to access data on a tape ending label after it is read or before it is written.

After execution of a USE procedure, control returns to the invoking routine, that is, to the statement following the READ that detected the error condition. In the case of blocked or unblocked magnetic tape input, the tape is positioned ready to read the next record as soon as the USE procedure is completed. For example, if you wish to print the records that cannot be read because of an unrecoverable parity error, a procedure can be included, as illustrated by example 2 in Appendix E.

Within a USE procedure, no statement can be executed which would cause the execution of a USE procedure that had previously been invoked but had not yet returned control to the invoking routine.

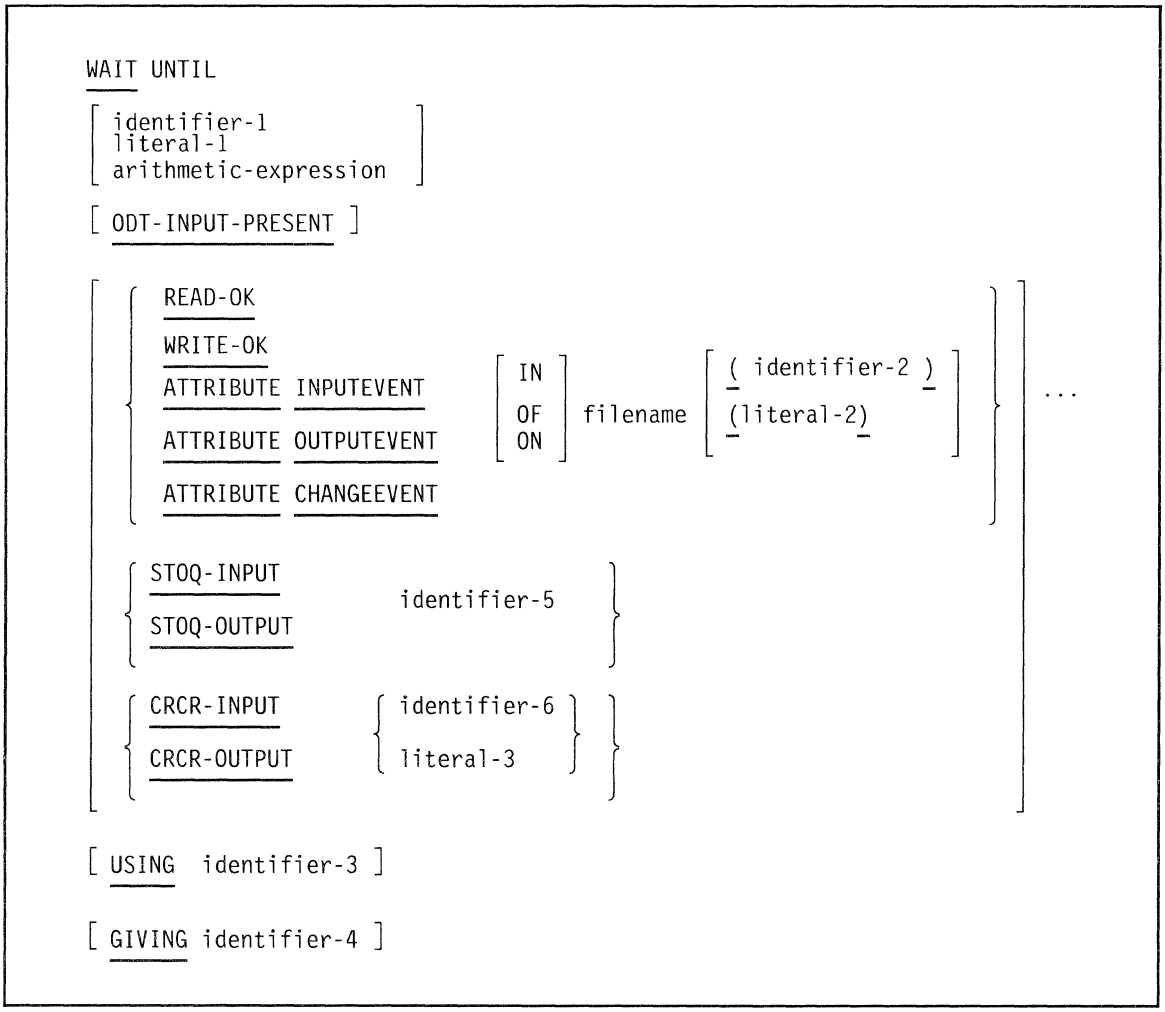
Within a USE procedure, there must be no reference to any nondeclarative procedures. Conversely, in the nondeclarative portion there must be no reference to procedure-names that appear in the declarative portion, except that PERFORM statements can refer to a USE statement or to the procedures associated with such a USE statement.

An example of a program that accesses tape labels is included in Appendix E.

WAIT Statement

The WAIT statement suspends the execution of the object program for a specified interval of time or until one or more conditions is true.

The following is the general format:



A WAIT statement with no options specified causes the program to be suspended for 86400 seconds (24 hours).

Not more than one arithmetic expression, literal-1, or identifier-1 can be specified. If specified, it must be the first item in the list. Identifier-1 and literal-1 must be numeric data-items. The value of the arithmetic expression, identifier-1, and literal-1 must not exceed 86400.

When arithmetic-expression, literal-1, or identifier-1 is specified, the execution of the object program is suspended for the number of seconds determined by the value of the specified item. If the value is negative, the number of seconds is

zero. If the value is zero, the program is not suspended and processing continues. The expression must reduce to an integer value between 1 and 86399. (Zero is equivalent to 1. Values greater than 86399 are treated as 86400, or one day.)

When ODT-INPUT-PRESENT is specified, the execution of the object program is suspended until a message is entered from the Operator Display Terminal (ODT).

READ-OK and ATTRIBUTE INPUTEVENT are synonymous. When either is specified, the program is suspended until there is at least one record to be read in the file (that is, CENSUS is greater than 1).

WRITE-OK and ATTRIBUTE OUTPUTEVENT are synonymous. When either is specified, the program is suspended until there is space in the file for at least one more record (that is, CENSUS is less than MAXCENSUS).

When ATTRIBUTE CHANGEEVENT is specified, the program is suspended until FILESTATE has changed.

The filename can refer to a PORT, DCP or ISC device. Identifier-2 or integer-2 specifies the subport of the file.

If any event in the list is true, the WAIT is terminated, and control is passed to the next executable statement.

If, during the initial scan of the events, no event is found to be true, program execution is suspended until any event in the list becomes true. At that time, the WAIT statement is terminated and control is passed to the next executable statement.

When USING is specified, the value of identifier-3 determines the event that should be tested first for a true condition. (The default, when USING is not specified, is that the first item in the list is always tested first. USING, then, can permit the programmer to specify a round-robin or other sequence of testing.)

When GIVING is specified, the data item specified by identifier-4 is set to the position that terminated the WAIT. The first item in the list is position number 1.

When STOQ-INPUT is specified, the program is suspended until a STOQUE entry is available. When STOQ-OUTPUT is specified, the program is suspended until space is available for storage of data. Identifier-5 must be an 01 level specifying the appropriate STOQUE Parameter Block.

When CRCR-INPUT is specified, the program is suspended until the sending program is ready to send the data. When CRCR-OUTPUT is specified, the program is suspended until the receiving program is ready to receive the data. Identifier-6 or literal-3 must have a length of 6 characters and specify the name of the program subject to the receive and send by way of core-to-core communication.

The Complex WAIT can be a more efficient method of causing program action when events occur than would be the case if explicit IF statements were used to test every possibility. Since WAIT suspends the program until one of the events

in the list has occurred, more time is available for other programs to run than would be the case if the program looped continuously through a series of IF statements, interrogating all of the events of interest.

When the program is reinstated from a WAIT, it must determine which event caused the wait to end. The GIVING clause contains a number indicating the position in the WAIT list of the item that ended the WAIT. The program must interrogate this item, and act upon its value.

Note: *If WAIT, WAIT integer-1, or WAIT identifier-1, is specified, a DOZE BCT, compatible with core-to-core, is generated. Otherwise, a COMPLEX WAIT BCT is generated.*

The following is an example of the use of WAIT.

```
WAIT UNTIL
      60
      ODT-INPUT-PRESENT
      ATTRIBUTE CHANGEEVENT ON PORTA
      ATTRIBUTE INPUTEVENT ON PORTB(PORTB-INDEX)
      ATTRIBUTE OUTPUTEVENT ON PORTC(2)
    GIVING WAKE-UP-CAUSE.

IF WAKE-UP-CAUSE = 1
    PERFORM    SIXTY-SECOND-STATUS
ELSE  IF WAKE-UP-CAUSE = 2
    PERFORM    PROCESS-OPERATOR-INPUT
ELSE  IF WAKE-UP-CAUSE = 3
    PERFORM    CHECK-PORTA-STATUS
ELSE  IF WAKE-UP-CAUSE = 4
    PERFORM    PROCESS-PORTB-INPUT
ELSE
    PERFORM    NEXT-FOR-PORTC.
```

This program waits until any of the following has occurred:

- Sixty seconds elapse since you entered the WAIT
- An ODT *AX* message is entered for the program
- A CHANGEEVENT happens on port file PORTA
- An INPUTEVENT happens for the subport of PORTB, which is designated by the subport index PORTB-INDEX
- An OUTPUTEVENT has occurred for subfile 2 of PORTC

When the WAIT ends, WAKE-UP-CAUSE contains a number from 1 to 5. This number identifies the event that caused the WAIT to end.

For DCP or ISC files, a WAIT ATTRIBUTE INPUTEVENT generates a DATA COMM WAIT BCT. A WAIT identifier-1 (or literal-1) followed by ATTRIBUTE INPUTEVENT generates a DATA COM WAIT BCT. In all other cases or combinations, the COMPLEX WAIT BCT is generated.

WRITE Statement

The WRITE statement releases a logical record for an output or input/output file. You can also use this statement for vertical positioning of lines within a logical page.

The following are the general formats:

Format 1

```
WRITE record-name [FROM identifier-1]
    [ { BEFORE } ADVANCING { { identifier-2 } [ LINE  ] }  ]
    [ { AFTER  }           { { integer      } [ LINES ] }  ]
    [                      { { mnemonic-name }           }  ]
    [                      { { PAGE          }           }  ]
    [ ; AT { END-OF-PAGE } imperative-statement ]
           { EOP        }
```

Format 2

```
WRITE record-name [WITH NO WAIT] [FROM identifier-1]
        [;INVALID KEY imperative-statement]
```

The record-name is the name of a logical record in the File Section of the Data Division and can be qualified.

The associated file must be open in the output, extend, or I/O mode at the time this statement is executed (see "OPEN Statement" earlier in this section).

Format 1 must be used if the ACTUAL KEY phrase is not specified in the FILE-CONTROL entry for the file associated with record-name. Format 2 must be used if the ACTUAL KEY phrase is specified in the File-Control entry for the file associated with the record-name.

In Format 2, the INVALID KEY phrase must be specified if an applicable use procedure is not specified for the associated file.

Executing the WRITE statement releases a logical record to the operating system. Executing a WRITE statement has no effect upon either the contents or accessibility of the record area. If the associated file is named in the SAME RECORD AREA, then the logical record is also available to the program as a record of other files referenced in the same SAME RECORD AREA clause.

If the ACTUAL KEY phrase is specified for a mass storage file whose access mode is sequential, then successfully executing a Format 2 WRITE statement updates the contents of the ACTUAL KEY data item to the ordinal number of the logical record written.

For a mass storage file whose access mode is random, executing a Format 2 WRITE statement releases the record area to the logical record of the file specified by the contents of the ACTUAL KEY data item.

For a mass storage file whose access mode is sequential, executing a Format 2 WRITE statement releases the record area to the next logical record in the file.

Note: *Records in a sequential file opened I/O cannot normally be replaced by a WRITE statement during an update operation. A READ-MODIFY-WRITE sequence would access logical record n, modify it, and write it into logical position n+1 in the file. The next READ would access logical record n+2 and so on. In order to modify a record n in place in a sequential file, the sequence READ-MODIFY-REWRITE must be used.*

The INVALID KEY condition exists under the following circumstances for a sequential file:

- For a mass storage file whose access mode is random, an INVALID KEY condition exists when the value of the ACTUAL KEY data item is less than 1 or greater than the ordinal number of the last logical record allowed for the file if the maximum logical size of the file is specified.
- For a mass storage file whose access mode is sequential, the INVALID KEY condition exists when a maximum logical size is specified for the file and no more logical records can be written.

The INVALID KEY condition exists under the following circumstances for a relative file:

- When the access mode is random or dynamic and the RELATIVE KEY data item specifies a record that already exists in the file
- When an attempt is made to write beyond the externally defined boundaries of the file

The INVALID KEY condition exists under the following circumstances for an indexed file:

- When the file is opened in the output or I/O mode and the value of the prime record key is equal to the value of the prime record key of a record already existing in the file

- When the file is opened in the output or I/O mode and the value of an alternate record key for which duplicates are not allowed equals the corresponding data item of a record already existing in the file
- When an attempt is made to write beyond the externally defined boundaries of the file

When the INVALID KEY condition is recognized, the execution of the WRITE statement is unsuccessful, the contents of the record area are unaffected, and the FILE STATUS data item of the file, if specified, is set to a value indicating an INVALID KEY condition.

Executing the WRITE statement updates the value of the FILE STATUS data item, if any, associated with the file. Executing the program proceeds unless character 1 of the FILE STATUS data item contains an F (see "File-Control Paragraph" in Section 4).

The maximum record size for a file is established when the file is created. It must not subsequently be changed.

The number of character positions on a mass storage device required to store a logical record in a file might or might not be equal to the number of character positions defined by the logical description of that record.

The current record pointer is unaffected by executing a WRITE statement.

Record-name and identifier-1 must not reference the same storage area.

The results of executing the WRITE statement with the FROM phrase is equivalent to executing the following statement:

```
MOVE identifier-1 TO record-name
```

This is according to the rules specified for the MOVE statement followed by the same WRITE statement without the FROM phrase. The contents of the record area before executing the implicit MOVE statement have no effect on executing this WRITE statement.

Both the ADVANCING phrase and the END-OF-PAGE phrase allow control of the vertical positioning of each line on a representation of a printed page. If the ADVANCING phrase is not used, automatic advancing is provided to act as if the user had specified AFTER ADVANCING 1 LINE. WRITE ... BEFORE ADVANCING is more efficient than WRITE ... AFTER ADVANCING. It is suggested, therefore, that programs writing printer files specify WRITE ... BEFORE ADVANCING rather than a simple WRITE. If the ADVANCING phrase is used, advancing is provided as follows:

- If identifier-2 is specified, the representation of the printed page is advanced the number of lines equal to the current value associated with identifier-2. This value of identifier-2 must not exceed 99. Identifier-2 must be the name of an elementary integer data item.
- If integer is specified, the representation of the printed page is advanced the

number of lines equal to the value of integer. The value of this integer must not exceed 99.

- If a mnemonic-name is specified, the representation of the printed page is advanced to the line number corresponding to the channel number.
- If the BEFORE phrase is used, the line is presented before the representation of the printed page is advanced, as outlined in the previous paragraphs.
- If the AFTER phrase is used, the line is presented after the representation of the printed page is advanced, as outlined in the previous paragraphs.
- If PAGE is specified, the record is presented on the logical page before or after (depending on the phrase used) the device is repositioned to the next logical page. If the record to be written is associated with a file whose file description entry contains a LINAGE clause, the repositioning is to the first line that can be written on the next logical page, as specified in the LINAGE clause. If the record to be written is associated with a file whose file description entry does not contain a LINAGE clause, the repositioning is to CHANNEL 1 or line 1 of the next logical page when appropriate for the hardware device.

  If page has no meaning in conjunction with a specific device, then advancing is provided to act as if the user had specified BEFORE or AFTER (depending on which was used) ADVANCING 1 LINE. If the logical end of the representation of the printed page is reached during the execution of a WRITE statement with the END-OF-PAGE phrase, the imperative-statement specified in the END-OF-PAGE phrase is executed. The logical end is specified in the LINAGE clause associated with record-name.

An end-of-page condition is reached whenever executing a given WRITE statement with the END-OF-PAGE phrase causes printing or spacing within the footing area of a page body. This condition occurs when executing such a WRITE statement causes the LINAGE-COUNTER to equal or exceed the value, if specified, in the FOOTING clause, that is, the value specified by integer-2 or data-name-2 of the LINAGE clause. In this case, first the WRITE statement and then the imperative statement in the END-OF-PAGE phrase is executed.

An automatic page overflow condition is reached whenever the execution of a given WRITE statement (with or without an END-OF-PAGE phrase) cannot be fully accommodated within the current page body. This situation occurs when a WRITE statement, if executed, would cause the LINAGE-COUNTER to exceed the total number of lines specified for a page, that is, the value specified by integer-1 or the data item referenced by data-name-1 of the LINAGE clause. In this case, the record is presented on the logical page before or after (depending on the phrase used) the device is repositioned to the first line that can be written on the next logical page, as specified in the LINAGE clause. The imperative statement in the END-OF-PAGE clause, if specified, is executed after the record is written and the device has been repositioned.

If you do not specify a FOOTING clause in the LINAGE clause, no end-of-page condition distinct from the page overflow condition is detected. In this case, the end-of-page condition and page overflow condition occur simultaneously.

If a FOOTING clause is specified in the LINAGE clause, that is, if integer-2 or data-name-2 of the LINAGE clause is specified but executing a given WRITE statement would cause LINAGE-COUNTER to simultaneously exceed the value of both the total number of lines allowed on a page (integer-2 or the data item referenced by data-name-2) and the footing area (integer-1 or the data item referenced by data-name-1), then the operation proceeds as if a footing area had not been specified.

When you specify mnemonic-name, it must be associated with a CHANNEL number. The mnemonic-name is defined in the SPECIAL-NAMES paragraph of the Environment Division.

When identifier-2 is used in the ADVANCING phrase, it must be the name of an elementary integer data item.

Integer or the value of data item referenced by identifier-2 can be 0.

If you specify the END-OF-PAGE phrase, you must also include the LINAGE clause in the file description entry for the associated file.

The words END-OF-PAGE and EOP are equivalent.

The ADVANCING mnemonic-name phrase cannot be specified when you write a record to a file whose file description entry contains the LINAGE clause.

When an attempt is made to write beyond the externally defined boundaries of a sequential file, an exception condition exists and the contents of the record area are unaffected. The following action takes place:

- The value of the FILE STATUS data item of the associated file, if any, is set to a value indicating a boundary violation.
- If a USE AFTER STANDARD EXCEPTION declarative is explicitly or implicitly specified for the file, that declarative procedure is then executed.
- If a USE AFTER STANDARD EXCEPTION declarative is not explicitly or implicitly specified for the file, the result is undefined.
- After recognizing an end-of-reel or an end-of-unit of an output file that is contained on more than one physical reel/unit, the WRITE statement performs the following operations:
  - The standard ending reel/unit label procedure
  - A reel/unit swap
  - The standard beginning reel/unit label procedure

The following apply specifically to relative files:

- When a file is opened in the output mode, records can be placed into the file by one of the following:
  - If the access mode is sequential, the WRITE statement releases a record to the mass storage control system. The first record has a relative record

number of 1, and subsequent records released have relative record numbers of 2, 3, 4, ... If the RELATIVE KEY data item has been specified in the file control entry for the associated file, the relative record number of the record just released is placed into the RELATIVE KEY data item during execution of the WRITE statement.

- If the access mode is random or dynamic prior to executing the WRITE statement, the value of the RELATIVE KEY data item must be initialized in the program with the relative record number to be associated with the record in the record area. That record is then released to the mass storage system.

- When a file is opened in the I/O mode and the access mode is random or dynamic, records are to be inserted in the associated file. The value of the RELATIVE KEY data item must be initialized by the program with the relative record number to be associated with the record in the record area. Execution of a WRITE statement then causes the contents of the record area to be released to the mass storage system.

The following apply to indexed files:

- Executing the WRITE statement releases the contents of the record area. The mass storage control system utilizes the contents of the record keys so that subsequent access of the record can be made based upon any of those specified record keys.
- The value of the prime record key must be unique within the records in the file.
- The data item specified as the prime record key must be set by the program to the desired value prior to executing the WRITE statement.
- If sequential access mode is specified for the file, records can be written in any order of prime record key values. The key file structure ensures that subsequent retrieval in sequential mode returns the records in ascending order of the prime record key.
- If random or dynamic access mode is specified, records can be written in any program-specified order.
- When the ALTERNATE RECORD KEY clause is specified in the FILE-CONTROL SELECT entry for an indexed file, the value of the alternate record key can be nonunique only if the DUPLICATES phrase is specified for that data item. In this case, storage of records is provided so that order of retrieval of records accessed sequentially, is the order in which they were written.

The following apply specifically to port files:

- The WITH NO WAIT phrase can be specified only for port files. Format 2 of the WRITE statement must be used for port files.
- A WRITE statement causes the program to wait until a buffer is available to store the record. For a port file, the possibility of this suspension is prevented by using the WITH NO WAIT phrase. A status key of 95 indicates that no buffer was available for the logical record.

- If an actual key is declared for a port file, the user is responsible for updating the ACTUAL KEY with an appropriate value. A WRITE statement passes the ACTUAL KEY to the system to indicate the desired subfile destination. If the ACTUAL KEY value is zero, a broadcast write is performed for which the data is sent to all opened subfiles of the port file.
- If no ACTUAL KEY is declared for a port file, the file must contain a single subfile to which the data is written.

Note: *If program A opens and writes into a port file but no other program opens the port file prior to program A closing the port file, all data is purged. The port block's memory is returned as available.*

DCP Files

The normal WRITE BCT is generated for the DCP function. You can declare a USE ON EXCEPTION routine in the DECLARATIVES section of the program to handle I/O errors. If the USE routine is specified, it is invoked when the MCP returns a type 29 result header for a WRITE. If the USE routine is not specified, you should check for a 29 result header immediately after each WRITE of the DCP file.

Section 7
File Attributes

General

File attributes allow a user program to request the compiler to generate code to directly manipulate or interrogate the file information block (FIB) and label areas of files within the user program.

The compiler does not verify the consistency of attribute values. Therefore, users must be aware that they are responsible for the logical correctness of file attributes.

No checking is done to ensure that the setting of attributes does not conflict with the operating system's expectations. For example, disk files require EXTMODE to be NONSTANDARD; however, if the file attribute KIND is changed to DISK, the attribute EXTMODE is not checked.

The code checks the file to ensure that it is closed before changing any attributes that require the file to be closed.

When a physical file is assigned to the logical file, attributes that are not relevant to the physical file are ignored. If, when the file is opened, any logical attribute is found to be invalid, the I/O subsystem gives the program a fatal run-time error.

Assigning a physical file to the logical file can change the attributes of the logical file. This change can be the result of operator intervention in file assignment, in label equation, or in contradictory attribute settings.

The values returned to attributes while a physical file is assigned to a logical file are the values in use by the I/O subsystem. The values can also be affected by system options.

Categories

A file attribute can be classified as belonging to one of three categories: alphanumeric, numeric, or mnemonic. The Type column of Table 7–1 contains either an A, N, or M to indicate the respective category of each attribute.

Alphanumeric Attributes

Where allowed in the syntax, an alphanumeric file attribute is like an elementary alphanumeric DISPLAY data item having a size equal to the maximum size allowed for the specified attribute. Its content is left-justified with space fill. Alphanumeric file attributes are allowed as operands in relation conditions and as sending operands in MOVE statements without a CORRESPONDING clause.

Numeric Attributes

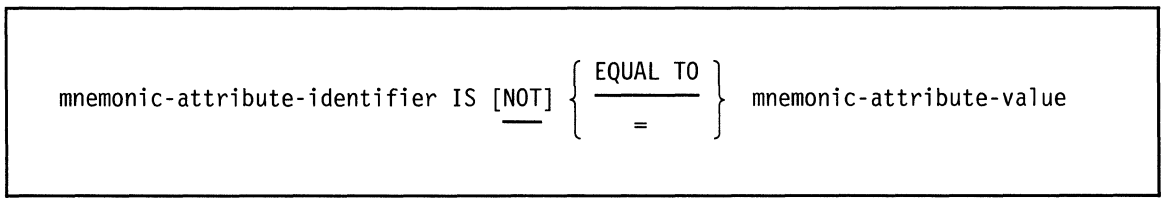
Where allowed in the syntax, a numeric file attribute is considered analogous to an elementary numeric DISPLAY data item that represents a signed integer with eight decimal digits. Numeric file attributes are allowed as sending operands in MOVE statements without a CORRESPONDING clause. Some numeric file attributes represent information that accounts for items such as the number of areas, blocks, or records within the file. These attributes are said to be "1 relative" in that their values specify the exact number of areas, blocks, or records in the file.

Mnemonic Attributes

Certain file attributes are associated with values that are best expressed as mnemonic names. Mnemonic file attributes can appear as the subject of a mnemonic attribute relation condition; that is, a valid mnemonic-attribute-value is associated with the object of a relation condition in which a mnemonic-attribute-identifier appears as the subject.

Mnemonic attribute relation conditions are allowed in any conditional expression.

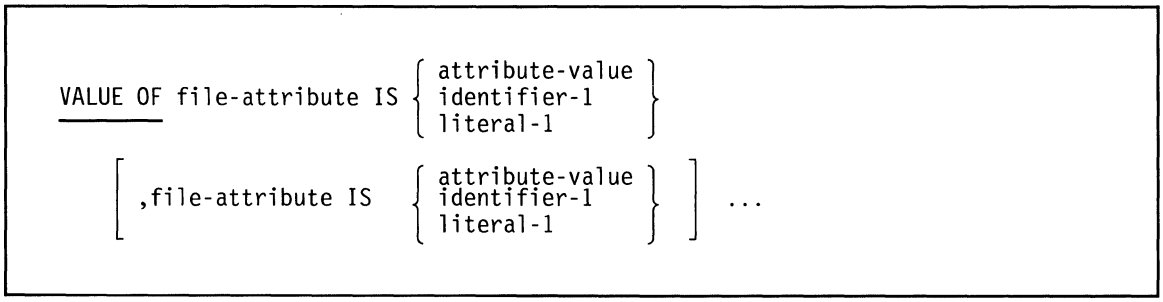
The general format of a mnemonic attribute relation condition is as follows:



Mnemonic attribute relation conditions cannot be abbreviated. The names for the mnemonic attribute values are system names, but not necessarily reserved words. File attributes having a Boolean nature are considered mnemonic attributes and are associated with the mnemonic attribute values TRUE and FALSE. When interrogated, these Boolean attributes return a value of zero for FALSE and one for TRUE.

References to File Attributes

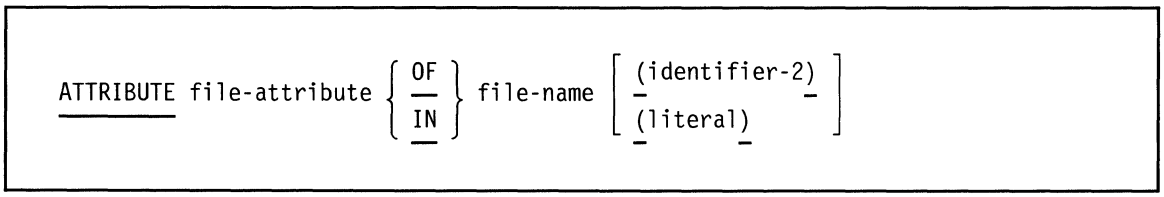
You can specify file attributes as part of the FD declaration, using the VALUE OF clause. The following is the syntax of this clause:



For further information on the VALUE OF clause, refer to "VALUE OF Clause" in Section 5.

In the Procedure Division of the program, a file attribute is referenced by the file attribute identifier-1 in a MOVE or IF statement.

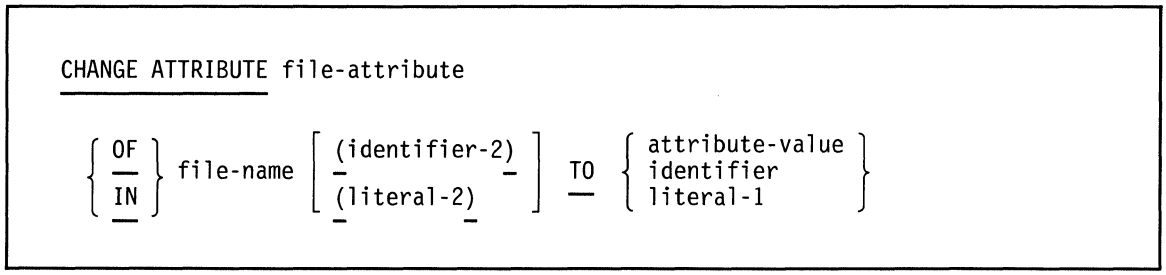
The following is the syntax of the file attribute identifier-1:



The file-attribute can be any valid attribute listed in column 1 of Table 7–1. The file-name is the file-id of the file to which the attribute applies.

Identifier-2 or literal-2 is a subport index, present only when file-name is a port file. The index specifies the subport to which the attribute applies.

You can modify or redefine file attributes in the Procedure Division of the program by using the CHANGE verb. The following is the syntax of this statement:



Identifier-1 or literal-1 applies only to a numeric or alphanumeric attribute. The attribute-value applies to a mnemonic attribute.

Identifier-2 or literal-2 is a subport index, present only when the file-name is a port file. The index specifies the subport to which the attribute applies.

For further information on the CHANGE statement, see "CHANGE Statement" in Section 6.

You can interrogate file attributes in the Procedure Division by using them as operands in relation conditions and as sending fields in MOVE statements without a CORRESPONDING clause. This usage varies with the category of the attribute (alphanumeric, numeric, or mnemonic). Usage is explained under "Categories" in this section.

Attributes and Values

Table 7–1 contains a list of all valid file attributes.

Column 1 is the file-attribute name. Column 2 lists the corresponding allowable attribute-value. Column 3 contains any hardware restrictions.

Column 4 contains either an RW indicating the attribute can be both read and written, an RO indicating the attribute can only be read, or a WO indicating the attribute can only be written.

Columns 5 and 6 indicate when the attribute can be read and/or written: *A* means the file must be assigned to the program, *C* means the file must be closed, *O* means the file must be open; *V* means variable; that is, the attribute can be read and/or written at any time.

The last column indicates the attribute category: *A* means alphanumeric, *N* means numeric, and *M* means mnemonic, which includes Boolean.

Table 7–1. File Attributes

File Attributes	Attribute Value	Valid Hardware	Read / Write	Read	Write	Type
ACCESSMODE	SEQUENTIAL RANDOM	Disk Diskpack	WO	--	C	M
AREAS	0 thru 99	Disk Diskpack	RW	V	C	N
AREASIZE	1 thru 99999999	Disk Diskpack	RW	V	C	N
BACKUPKIND	TAPE DISK DISKPACK	Printer	RW	V	C	M
BACKUPPERMITTED	DONTBACKUP DONTCARE MUSTBACKUP	Printer Punch	WO	--	C	M
BLOCK	0 thru 99999999	General	RO	V	--	N
BLOCKSIZE	2 thru 999998	General	RW	V	C	N
BLOCKSTRUCTURE	FIXED VARIABLE	General	RW	V	C	M
BUFFERS	0 thru 9	General	RW	V	C	N
CREATIONDATE	0 thru 99366	Tape	RO	A	--	N
CURRENTBLOCK	4 thru 999998	General	RO	V	--	N

continued

Table 7–1. File Attributes (cont.)

File Attributes	Attribute Value	Valid Hardware	Read / Write	Read	Write	Type
CYLINDERMODE	TRUE FALSE	Diskpack	RW	V	C	M
DENSITY	BPI556 BPI800 BPI1600 BPI6250 BPI38000	Tape	RW	V	C	M
DIRECTION	FORWARD REVERSE	Tape	RW	V	C	M
EXTMODE	NONSTANDARD BINARY EBCDIC	General	RW	V	C	M
FAMILYNAME	String(6)	Diskpack Tape	RW	V	C	A
FILENAME	String(6)	General	RW	V	C	A
FORCEIO	TRUE FALSE	Disk Diskpack	RW	V	V	M
FORMID	String (6)	Printer Punch	RW	V	C	A
HOSTNAME	String (17)	Disk Diskpack Port	RW	V	C	A
KIND	DCP DISK DISKPACK ISC PAPERREADER PAPERPUNCH PRINTER PORT PUNCH READER TAPE TAPE7 TAPE9 TAPEPE TAPEGCR TAPEMTC	General	RW	V	C	M
LABELKIND	INSTALLATION MSSTANDARD OMITTED MSUSASII	Tape	RW	A	C	M

continued

Table 7–1. File Attributes (cont.)

File Attributes	Attribute Value	Valid Hardware	Read / Write	Read	Write	Type
LABELSIZE	40 thru 9996	Tape	RW	V	V	N
LASTRECORD	0 thru 99999999	Disk Diskpack	RO	A	--	N
MAXRECSIZE	4 thru 99996	General	RW	V	C	N
MYUSE	IN OUT IO	General	RW	V	C	M
OPEN	TRUE FALSE	General	RO	V	--	B
OPTIONAL	TRUE FALSE	General	RW	V	C	M
PARITY	ODD EVEN	Tape	RW	V	C	M
PROTECTION	TEMPORARY ABNORMALSAVE SAVE	Disk	RW	A	C	M
SAVEFACTOR	1 thru 999	Tape	RW	V	C	N
SECURITYFAMILY	String(6)	Disk Diskpack	WO	--	C	A
SECURITYGUARD	String(6)	Disk Diskpack	WO	--	C	A
SECURITYTYPE	PRIVATE GUARDED PUBLIC	Disk Diskpack	WO	--	C	M
SECURITYUSE	SECURED IN OUT IO	Disk Diskpack	WO	--	C	M
SENSITIVEDATA	TRUE FALSE	Disk Diskpack	WO	--	C	M
SERIALNO	String(5)	Tape	RO	V	--	A
SHARED	TRUE FALSE	Disk Diskpack	RW	V	C	M
SINGLEUNIT	TRUE FALSE	Diskpack	RW	V	C	M

continued

Table 7–1. File Attributes (cont.)

File Attributes	Attribute Value	Valid Hardware	Read / Write	Read	Write	Type
SUBSYSTEM	DSK1 DSK5 DSK2 DSK6 DSK3 DSK7 DSK4 DSK8 DEFAULT	Disk	RW	V	V	M
SYNCHRONIZE	TRUE FALSE	Tape	RW	V	C	M
UNIQUENAME	PROCESSOR WORK NULL	General	WO	--	C	M
UNITNO	String(3)	General (NOT Disk)	RO	A	--	A
VOLUMEINDEX	0 thru 999	Tape	RW	A	C	N

The file attributes are discussed individually on the following pages.

ACCESSMODE

The attribute ACCESSMODE specifies the manner in which records are to be accessed. SEQUENTIAL is the default and indicates that records are accessed in an established sequence. RANDOM indicates that records are accessed by the relative record number specified by a record key.

AREAS

The value of the attribute AREAS is the maximum number of areas a disk or diskpack file can contain. The AREAS attribute can be set only when the file is closed. The default is 0, which the MCP interprets as 100.

For all other disk and diskpack files, when opened INPUT, the MCP reads up the AREAS attribute as found in the disk or diskpack file header. The MCP writes this value into the program FIB. This value is not changed if the file is subsequently opened OUTPUT. This attribute cannot be changed for indexed files.

AREASIZE

The value of the attribute AREASIZE is the number of records in an area of a disk file. Set this attribute only before creating the file. When you create a new disk or diskpack file, the value of AREASIZE is adjusted so that it is divisible by the block size (refer to the BLOCK CONTAINS clause, part of the Data Division

FD entry). If AREASIZE is zero, a fatal run-time error occurs. The default is 1000.

When a disk or pack file is opened INPUT,the MCP reads up the AREASIZE attribute as found in the disk or diskpack file header and writes this value into the program FIB. This value is not changed by a subsequent open OUTPUT of this file. You cannot change this attribute for INDEXED organization.

BACKUPKIND

This attribute specifies the backup device type for a printer output file. (Refer also to the BACKUPPERMITTED file attribute.)

BACKUPPERMITTED

This attribute indicates whether an output printer or punch file can be directed to a backup file. The default DONTCARE means that whether or not the file goes to backup depends entirely upon system options. The choice DONTBACKUP prevents the file from going to backup automatically. If a printer or punch device is not available, the file can be manually assigned to a backup by use of the system command OU. The choice MUSTBACKUP forces the file to a backup device. You can specify the backup device type for a printer output file by using the file attribute BACKUPKIND. The backup device type for a punch file is DISK.

BLOCK

The BLOCK attribute returns the number of the logical block referenced in the last I/O statement. The first block in the file is block one (1). Before an I/O has been performed on the file or after the file is closed, BLOCK returns to zero (0). Sequential files with multiple buffers cause buffers to be filled at file OPEN, which causes the BLOCK value to be greater than zero (0) before an I/O request.

Note: *If a sequential file has multiple buffers, the buffers are filled when the file is opened INPUT. Doing so causes BLOCK to return a value greater than zero before reading a multibuffered sequential file.*

BLOCK returns a value between 0 and 99999999, and can be interrogated anytime. (BLOCK is a read-write attribute.)

BLOCKSIZE

BLOCKSIZE is the actual size of a block (buffer) of fixed-length records, expressed as the number of digits. BLOCKSIZE must be a multiple of the size of the largest record declared for the file.

BLOCKSIZE is a value between 2 and 999998 (40,000 for indexed and relative files). It can be interrogated at any time, but can be modified only when the file is closed.

Note: A program must not attempt to modify BLOCKSIZE to a value larger than that declared in the FD for the file.

BLOCKSTRUCTURE

This attribute specifies the format of the records and the structure of the file. FIXED indicates blocked or unblocked fixed-length records. VARIABLE indicates variable-length records; in this case, the record length is contained in the first four characters of the record. Only files with KIND = TAPE can have BLOCKSTRUCTURE = VARIABLE. The default is FIXED.

The blocking technique used with variable-length blocked files is as follows:

For work area access files, a physical write is initiated for the block if the record to be written plus the offset into the block is greater than BLOCKSIZE. The record to be written becomes the first record of the next block.

For buffer only files, the block is written when there is no longer sufficient space in the block for the longest record. If the logical file is assigned to a peripheral unit (KIND) that allows variable-length blocks (tape files), then only the part of the block that is valid is written.

BUFFERS

The BUFFERS attribute specifies the number of alternate I/O buffers assigned to a file; the maximum number of buffers assigned is 10. However, the actual number of buffers assigned is FIBALT + 1. This attribute cannot be changed for indexed and relative files.

There must be one buffer status word for each buffer declared.

Note: A program must not attempt to increase BUFFERS to a value greater than that specified in the RESERVE clause of the file's SELECT statement.

CREATIONDATE

This attribute returns the creation date of a tape file. The tape must have a V Series system standard label. The value of the CREATIONDATE attribute is returned as an integer in the Julian form YYDDD, where YY is the year and DDD is the day. When a tape file is created, TODAY-DATE is always used as the creation date of the file.

CURRENTBLOCK

The CURRENTBLOCK attribute returns the size in digits of the block currently in use. Normally, this value is the same as the value of the BLOCKSIZE attribute. CURRENTBLOCK is utilized when the system encounters a short block for a tape file.

CYLINDERMODE

When this attribute is TRUE, the areas of a diskpack file are assigned so that the areas are contained in the minimum possible number of cylinders. The default value is FALSE.

DENSITY

The DENSITY attribute specifies the recording density of a magnetic tape file; valid options are the following:

BPI556
BPI800
BPI1600
BPI6250
BPI38000

The following rules apply to the DENSITY attribute:

- BPI556 is not valid for 9-track tape.
- BPI38000 is valid only if the KIND attribute of the tape is declared as TAPEMTC.
- The default DENSITY value for tape files declared as KIND=TAPE is BPI1600.

The following exceptions apply to the DENSITY attribute:

- When a multifile tape is created, the density of the first file determines the density for all remaining files.
- When a multireel tape is created, the density is constant for all reels.

DIRECTION

The DIRECTION attribute indicates the direction in which records are accessed from a file. Valid entries are FORWARD and REVERSE, with a default value of FORWARD.

Only tape files with BLOCKSTRUCTURE = FIXED can be read in a REVERSE direction.

EXTMODE

This attribute specifies the external or physical character size (mode) of the records in a file. NONSTANDARD is required for disk and diskpack files. For punch files, the file is to be punched in BCL. BINARY applies only to punch files and means the file is to be punched in binary form (low order 6 bits of even bytes punched in the top rows of the card, low order 6 bits of odd bytes punched in the bottom rows). This punch file then cannot go to backup. EBCDIC is the default value for both card and punch files.

This EXTMODE attribute can be overridden by the physical mode of a permanent file or unit type.

FAMILYNAME

This attribute specifies the multi-file-id to be used for assignment to a multifile tape, or the diskpack family to be used for file assignment. The attribute value can be any string of up to six characters that constitute a valid multi-file-id.

FILENAME

This attribute is an external file name. It is used to associate a logical file with a physical or permanent file. The attribute value can be any string of up to six characters that constitute a valid file-id. The default FILENAME for the file is the value of the file's internal name assigned by the compiler.

The names of indexed files must be unique in the first four characters.

When an indexed file is created, the access routines create, separate from the data file, a key file to hold the RECORD KEY information and a file for each ALTERNATE RECORD KEY named for the indexed file. The names of these files are created from the first four characters of the name of the indexed file; two numeric characters, beginning with 01 for the first key file, are appended to make a 6-character name. Two indexed file names with the same first four characters would therefore lead to duplicate key file names.

COBOL allows multiple file tape for sequential I/0. If the VALUE OF FAMILYNAME clause is not present, the default can be overridden by CHANGE ATTRIBUTE FAMILYNAME or file equate. When the final and intermediate files are opened OUTPUT, the compiler generates the OPEN with NO REWIND option. The NO REWIND option is also generated when the first and intermediate files are closed. Doing so permits files to be created properly and accessed without explicit NO REWIND's; however, the access is more efficient with user specified NO REWIND statements.

FORCEIO

This attribute, when TRUE, specifies that the file's buffers are not to be examined for the block in which the requested record resides; a physical read must be performed. The default value is FALSE. This attribute has meaning only when ACCESSMODE = RANDOM.

For indexed and relative files, the meaning of FORCEIO is quite different. When set to TRUE, all changed blocks (data and key blocks) are written to their respective files at the end of each update operation. Also, each logical read causes a physical read on the file even if the desired block is in the buffer. The default value is FALSE, and this attribute is meaningful regardless of ACCESSMODE.

FORMID

The printer file attribute, FORMID = <string>, where the string is a 6-character literal or data-name, specifies the following:

- Special forms are requested when the file is physically printed.
- The value in the FORMID string is used as the multi-file id of the print file so that the listing can be directed to a particular printer.
- The special forms flag is reset if the value in the FORMID string is null or blank (represented by a period and spaces respectively).

When the file is physically printed or punched (either by the original creating program or by a subsequent printer or punch backup processing utility), the file is directed to the printer or punch whose Unit-ID matches the multifile-ID. If no unit with that name exists, a message is displayed at the ODT, notifying the operator that special forms are required. The program is then suspended until the operator enters the FM keyboard command to direct the file to a particular unit.

If the value following FORMID is blank or null, the special forms bit (FIBSPF) is reset.

HOSTNAME

The file attribute HOSTNAME can be associated with any disk or pack file. HOSTNAME specifies the name of the processor on which the file resides. If the HOSTNAME is the name of a processor other than the processor at which the COBOL program is executing, then both processors must be in a BNA network. A hostname is specified in the coldstart/warmstart HOSTNAME initialization parameter or through the HN system command.

In order to access a file that resides at a remote host, the COBOL program needs only to specify the HOSTNAME of the file (and to meet system security requirements, if applicable, at the remote host).

KIND

This attribute indicates the type of device associated with the logical file. Each logical file has exactly one value for KIND. The values include the following:

Table 7–2. KIND Attribute Values

KIND Attribute Value	Device
DCP	Data communications processor
DISK	100-byte random access storage peripheral
DISKPACK	180-byte random access storage peripheral
ISC	Inter-system control device
PAPERREADER	Paper tape reader
PAPERPUNCH	Paper tape punch
PORT	Port file
PRINTER	Any type of line printer
PUNCH	80-column card punch
READER	80-column card reader
TAPE	Any magnetic tape peripheral
TAPE7	7-track magnetic tape drive
TAPE9	9-track magnetic tape drive
TAPEPE	9-track phase encoded magnetic tape drive
TAPEGCR	9-track group character recognition magnetic tape drive
TAPEMTC	Magnetic tape cartridge device (GCR mode)

Tape Considerations

- When the KIND attribute is interrogated, hardware types TAPE, TAPE7, TAPE9, TAPEPE, TAPEMTC, and TAPEGCR return a value of TAPE.
- Tape drives that use the Small Computer Systems Interface (SCSI) can have only KIND values of TAPEGCR, TAPEPE or TAPE9.

LABELKIND

This attribute describes the kind of tape label desired. INSTALLATION indicates an installation label is to be used; MSSTANDARD indicates the standard V Series System label; OMITTED indicates label records are to be omitted; and MSUSASII indicates a V Series System USASII label is desired. MSSTANDARD is the default.

WARNING

All labels are not the same length. Once the file has been declared in the FD with a certain LABELKIND, changing this attribute in the Procedure Division to a larger label causes corruption of the data following the original label area.

LABELSIZE

This attribute allows you to define the size of the label area, expressed in digits, without a LABEL RECORDS clause. It also handles the variable number of label records allowed with INSTALLATION labels by updating the FIBLAE field in the FIB.

WARNING

Modifying the LABELSIZE value to be greater than that declared at compile time corrupts the data following the original label area.

LASTRECORD

The attribute LASTRECORD returns the record number of the last record in the physical file. LASTRECORD cannot be correct during periods when the file is being expanded, because the end-of-file calculations are made only when necessary, as for example, during the transition from writing to reading.

If the file is created by the current program, the file must be closed with RELEASE (or DISMISS) and reopened in order to obtain accurate results for LASTRECORD.

For indexed and relative files, LASTRECORD returns the last block number that contains records.

MAXRECSIZE

This attribute specifies the maximum size in digits of records in the logical file. MAXRECSIZE can be assigned a value between 4 (28 for tape files) and 99996 (39996 for files with workareas).

The attributes MAXRECSIZE, BLOCKSTRUCTURE, and KIND are closely related. MAXRECSIZE must be less than or equal to BLOCKSIZE. If BLOCKSTRUCTURE = FIXED, then BLOCKSIZE must be a multiple of MAXRECSIZE.

MYUSE

The MYUSE attribute specifies file usage; IO is input and output, IN is input only, and OUT is output only. The default value is IN.

OPEN

The OPEN attribute indicates whether or not a file is OPEN (TRUE or FALSE respectively).

OPTIONAL

This attribute indicates whether or not (TRUE or FALSE) the assignment of a permanent file is optional. If a permanent file is not present when the file is opened, a NO FILE message is sent to the ODT, and the program is suspended. If OPTIONAL = TRUE, the operator can respond with an OF system command and the program proceeds without a physical file being assigned to the logical file. Nonpresent files with attribute OPTIONAL = TRUE can be INPUT files only. Any attempted access results in end-of-file action. Writing to such a file is not allowed. Doing so proves fatal during program execution. The default value for the attribute OPTIONAL is FALSE.

PARITY

This attribute indicates tape file parity, with values as follows:

ODD - Binary or standard parity

EVEN - Alpha, or nonstandard parity

The default value is ODD. EVEN parity is valid only for 7-track magnetic tape files.

PROTECTION

This attribute indicates the amount of extra effort desired to preserve a file in case of a program failure (note that nothing helps in case of a system failure). The default value TEMPORARY means that a new file is purged when the program is discontinued unless the file is explicitly closed with an overriding CLOSE statement. If the disk file is locked or crunched, an entry is then made in the directory and the file is saved. If PROTECTION = ABNORMALSAVE, an entry is made in the directory if the program terminates abnormally while the file is open. When PROTECTION is set to SAVE, the default close type is RELEASE.

SAVEFACTOR

This attribute indicates the expiration date of a tape file in terms of the number of days past the creation date. The system does not purge a tape whose SAVEFACTOR has expired unless a scratch tape is required, at which point any tape whose SAVEFACTOR has expired is considered available for assignment.

SECURITYFAMILY

This attribute specifies the multi-file-id of the guard file if SECURITYTYPE is GUARDED. If SECURITYFAMILY is DISK, the guard file is assumed to reside on 100-byte disk media, the default. SECURITYFAMILY can be set only with the VALUE OF clause.

SECURITYGUARD

This attribute specifies the file-id of the guard file if SECURITYTYPE is GUARDED. The default file-id is spaces. It can be changed only with a VALUE OF clause.

SECURITYTYPE

This attribute specifies who, other than the creator of the file or a privileged user, can access the file.

The default value is that assigned to the usercode/password combination under which the program creating the file is run. The following are the possible values:

- PRIVATE - Only users running under the usercode/password of the creator of the file can access it, in the manner specified by the SECURITYUSE attribute.
- GUARDED - Another file identified by the SECURITYGUARD attribute must be accessed before a user can be given permission to use the file, in the manner specified by an entry in the guard file. The guard file includes a default SECURITYUSE for those usercodes not listed in the guard file.

  If the SECURITYGUARD attribute is missing or the specified guard file is not present, then PRIVATE security instead of GUARDED is invoked.

  The guard file can itself be guarded by security.
- PUBLIC - All users can access the file, in the manner specified by the SECURITYUSE attribute.

This attribute can be set only with the VALUE OF clause.

For further information on file security and system access control, refer to the *V Series System Software Security Installation and Operations Reference Manual.*

SECURITYUSE

This attribute specifies the manner in which a PRIVATE or PUBLIC file can be accessed. The default value is IO and can be changed only with the VALUE OF clause. The SECURITYUSE attribute is taken from the guard file if SECURITYTYPE is GUARDED. The following are the values the attribute can have:

- IN - Read-only access is allowed to the file.
- IO - Read/write access is allowed to the file.
- OUT - Write-only access is allowed to the file.
- SECURED - The file is a program and can be executed only if it passes the MCP requirements for a program. No maintenance to the file is permitted.

SENSITIVEDATA

This attribute, when set to TRUE, specifies that, when the file is scheduled to be removed (for example, by a CLOSE PURGE or an MCP REMOVE command), all records of the file are to be overwritten with a random data pattern before the file segments are returned to the available disk pool. SENSITIVEDATA can be set only with the VALUE OF clause and defaults to FALSE.

SERIALNO

This attribute can only be read (interrogated). It then returns the serial number of the labeled tape to which the logical file is assigned. The serial number of a tape, which is established (written) by the AC or SN system command, is a numeric string of up to five characters, right-justified in a field of zeroes. A MSUSASI labeled tape file must be opened before SERIALNO interrogation.

SHARED

For sequential files, this attribute, when set to TRUE, specifies that the file is shared between multiple processors. If a file is declared as SHARED, the access mode must be RANDOM.

This attribute is not allowed for relative files.

For indexed files, this attribute, when set to TRUE, specifies that the file can be accessed by multiple users in either input/output (I/O) or read-only mode. Access is through the shared ISAM handler. This attribute can be declared only in the VALUE OF clause.

SINGLEUNIT

This attribute indicates whether or not (TRUE or FALSE) continuation areas for a diskpack file are to be allocated on the same device as the base area is. The default value of this attribute is FALSE. When this attribute is set TRUE and space is unavailable on the original unit of allocation, the operator receives a notification. The operator can then respond by making more space available on the unit or by using the OU system command.

SUBSYSTEM

This attribute indicates the subsystem on which the physical file is to be located, as in the following:

- DSK1 - assign to subsystem 1
- DSK2 - assign to subsystem 2
- DSK3 - assign to subsystem 3, and so forth up to DSK8
- DEFAULT - assign to the default subsystem.

SYNCHRONIZE

This attribute indicates buffered or unbuffered output operations on magnetic tape devices. SYNCHRONIZE is only used with magnetic tape cartridge devices (KIND = TAPEMTC) or with magnetic tape drives that use the Small Computer Systems Interface (SCSI).

When SYNCHRONIZE is TRUE, the tape device performs unbuffered output operations. Data is written on or read from the tape at the time of each output request. This mode is frequently used when programs are creating backup or audit files.

When SYNCHRONIZE is FALSE, the tape device performs buffered output operations. Data is held in the controller of the device until a sufficient amount accumulates. When enough data accumulates, then the data is written to the tape.

The default value of SYNCHRONIZE is FALSE.

UNIQUENAME

This attribute specifies that the file-id assigned to this file is to have a measure of uniqueness on this particular execution. PROCESSOR indicates that the processor number of the system that the program was executed on is to be placed in the fifth character position of the file-id. WORK indicates that in addition to the processor action, the program's mix number is to be placed in the second and third character positions of the file-id. NULL is the default setting; it specifies no unique name. If you change UNIQUENAME from WORK or PROCESSOR to NULL, the file name is not changed, but can be if you use the FILENAME attribute.

UNITNO

This attribute can only be read (interrogated). It returns a 3-character number string, which is the device specifier to which the file is assigned. This attribute applies to all hardware devices except disk.

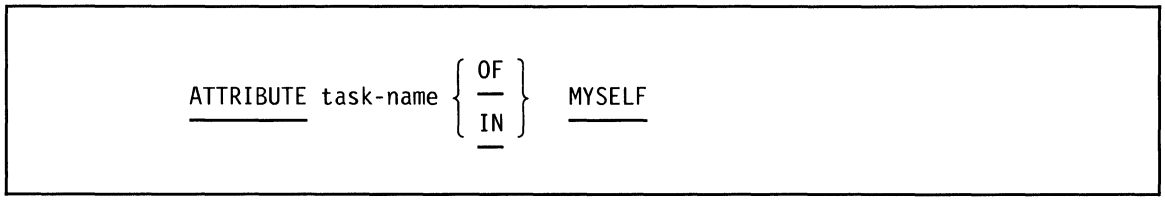
VOLUMEINDEX

This attribute specifies the reel number of a tape file. It is used along with the attributes KIND, FILENAME, and FAMILYNAME. Setting this attribute to a required tape reel number in a tape file label causes the operating system to start searching for the file on that reel of the multifile tape.

Task Attributes

Task attributes allow a task or program to request or modify certain of its characteristics. This form of system-dependent function is limited to use as the sending operand of a MOVE statement (without the CORRESPONDING clause) or as part of an IF statement.

The general format for task attributes is:



The following task attributes are available and perform the described operations.

ACCUMIOTIME

This task attribute is numeric and read only. It obtains the amount of I/O resource time the calling program has currently accumulated. The resulting value is a seven-digit unsigned numeric integer representing time in seconds.

ACCUMPROCTIME

This task attribute is numeric and read only. It indicates the amount of processor time the calling program has accumulated. The resulting value is a seven-digit unsigned numeric integer representing time in seconds.

MIXNUMBER

This task attribute is numeric and read only. It obtains the mix number of the calling program. The result is a four-digit numeric integer value.

PRIMARYSTORAGE

This task attribute is numeric and read only. It obtains the amount of primary storage assigned to the calling program. The result is a six-digit numeric value that indicates memory assigned in digits. If the calling program has data in HIGH MEMORY, the result is a seven-digit numeric value indicating memory assigned in digits.

Port and Subport File Attributes

You can set port attributes in the FD of a port file; but you can set subport attributes only in the Procedure Division.

Table 7-3 lists the attributes applicable to ports and subports. Default values , where they exist, are denoted by an asterisk (*). Also shown are codes to indicate whether the attribute is read-only (RO) or read-write (RW); whether the attribute can be read/written at any time (V) or only when the file is open (O) or closed (C); and whether the attribute value is mnemonic (M), numeric (N), or alphanumeric (A).

Table 7–3. Port and Subport Attribute Summary

File Attribute	Attribute Value (Default = *)	Valid Hardware	Read/ Write	Read	Write	Type
BLOCKSTRUCTURE	FIXED* EXTERNAL	Port	RW	V	C	M
CENSUS	0–999999* 0–9999*	Port Subport	RO RO	V V	- -	N N
CHANGEDSUBFILE	Subport index 0–9999*	Port	RO	V	-	N
CHANGEEVENT	Event (FALSE*)	Port Subport	RO RO	V V	- -	M M
COMPRESSION	Boolean (FALSE*)	Subport	RW	V	O	M
COMPRESSIONPOSSIBLE	Boolean (FALSE*)	Subport	RO	V	-	M
CURRENTRECORD	2 thru 19998 bytes	Subport	RO	V	-	N
DIALOGPROTOCOLLEVEL	0*–255	Subport	RO	V	-	N
FILESTATE	CLOSED* AWAITINGHOST OFFERED OPENED SHUTTINGDOWN BLOCKED CLOSEPENDING DEACTIVATIONPENDING DEACTIVATED	Subport	RO	V	-	M
HOSTNAME	String (17) (Local hostname*)	Disk Diskpack Subport	RW	V	C	A
INPUTEVENT	Event (FALSE*)	Port subport	RO	V	-	M

continued

Table 7–3. Port and Subport Attribute Summary (cont.)

File Attribute	Attribute Value (Default = *)	Valid Hardware	Read/ Write	Read	Write	Type
INTNAME	String (17)	Port	RW	V	C	A
LASTSUBFILE	Subport index 0*–99	Port	RO	V	-	N
MAXCENSUS	0–9999 (20*)	Subport	RW	V	V	N
MAXRECSIZE	0 thru 20000 bytes	Port Subport	RW RO	V V	C -	N N
MAXSUBFILES	1* thru 9999	Port	RW	V	C	N
MYHOSTNAME	String (17) (Local hostname*)	Port	RO	V	-	A
MYNAME	String (100) (NULL*)	Port	RW	V	C	A
OUTPUTEVENT	Event (FALSE*)	Subport	RO	V	-	M
SUBFILEERROR	NOERROR* DISCONNECTED DATALOST NOBUFFER NOFILEFOUND UNREACHABLEHOST	Subport	RO	V	-	M
TITLE	String (17) (INTNAME*)	Port	RW	V	C	A
YOURNAME	String (100) (NULL*)	Subport	RW	V	C	A
YOURUSERCODE	String (17) (User program usercode*)	Subport	RW	V	C	A

The following pages describe the port and subport file attributes.

Note that some string attributes (HOSTNAME, MYNAME, YOURNAME, and YOURUSERCODE) can be null. A null attribute is one whose value is a period (.). The period has the meaning *null* only for the attributes HOSTNAME, MYNAME, YOURNAME, and YOURUSERCODE; setting any other string attribute to period has meaning only to the user.

BLOCKSTRUCTURE

This attribute can contain either of the values FIXED or EXTERNAL for port files. The default is FIXED. The BLOCKSTRUCTURE attribute is meaningful only for read operations. When the value is FIXED, the user's buffer is blank-filled from the end of the data to the end of the buffer if the length of data received is less than the buffer size. If the value is EXTERNAL, only the data received is put into the user's buffer. You can determine the actual length of the data placed there by interrogating the CURRENTRECORD attribute.

CENSUS

As a port file attribute, CENSUS contains the total number of incoming messages queued from all subport files. As a subport attribute, CENSUS contains the number of messages queued for the specified subport. This attribute is read-only.

CHANGEDSUBFILE

This attribute is a read-only attribute and returns the subport index of an arbitrary subport whose CHANGEEVENT is *caused*.

CHANGEEVENT

For a port file the value of CHANGEEVENT is set to *caused* when any subport's CHANGEEVENT has the value *caused*. The system resets the CHANGEEVENT for the port after it has reset all of the subport CHANGEEVENTs. A subport CHANGEEVENT is set whenever the value of FILESTATE changes. It is reset as a result of interrogating the FILESTATE attribute.

COMPRESSION

This attribute causes the data sent between subport files to be compressed. Support for the compression feature is negotiated at subport open time. Setting the value of the COMPRESSION attribute to TRUE while the subport is open is effective only if both hosts involved in the subport dialog support the compression feature. (If they do not support compression, the value of COMPRESSION is FALSE even after it has been set to TRUE.) If compression is supported, you can selectively compress records by changing the value of the COMPRESSION attribute.

COMPRESSIONPOSSIBLE

This attribute indicates whether or not the local user can have the subport compression set.

CURRENTRECORD

This attribute is used only for subport files. CURRENTRECORD returns the length of the last record read when BLOCKSTRUCTURE is equal to EXTERNAL.

DIALOGPROTOCOLLEVEL

This attribute is read-only and it indicates the version of subport-to-subport protocol to be used.

FILESTATE

This attribute can have the following mnemonic values:

CLOSED
AWAITINGHOST
OFFERED
OPENED
SHUTTINGDOWN
BLOCKED
CLOSEPENDING
DEACTIVATIONPENDING
DEACTIVATED

For further information on FILESTATE values, refer to the *BNA Architectural Description Reference Manual, Volume 1.*

HOSTNAME

HOSTNAME is the name of the host processor with which this logical file is communicating. For a standard physical file, such as disk, this is the name of the host at which the file actually resides. For a port file, this is the name of the host at which resides the cooperating process with which this process intends to communicate using this subport.

INPUTEVENT

If accessed as a port attribute, INPUTEVENT returns *happened* if the port attribute CENSUS contains a value greater than zero. If accessed as a subport attribute, INPUTEVENT returns *happened* if the CENSUS subport contains a value greater than zero.

In a COBOL WAIT statement, ATTRIBUTE INPUTEVENT is equivalent to READ-OK.

INTNAME

The INTNAME of a port file is the name by which the port file is declared and referenced by the program that is using it.

LASTSUBFILE

This attribute contains the subport index of the last subport that was used in an I/O operation on the port file.

MAXCENSUS

This attribute specifies the number of input messages that can be queued for a subport before a cooperating subport is given a NO BUFFER AVAILABLE condition (the cooperating subport's SUBFILEERROR = NOBUFFER).

MAXRECSIZE

As a port attribute, MAXRECSIZE is set to indicate the maximum message text size for the port file. As a subport attribute, MAXRECSIZE returns the actual message text size for the subport. The value of the subport MAXRECSIZE is negotiated when the subport is being opened. The value can be different for each subport. The smaller of the two port files's MAXRECSIZE becomes the subport's MAXRECSIZE (at open time).

MAXSUBFILES

This attribute defines the maximum number of open subports a port file can have. When no value is specified for MAXSUBFILES, the default is 1.

MYHOSTNAME

This attribute contains the name of the local host.

MYNAME

This attribute identifies a port during the subport matching process.

OUTPUTEVENT

This attribute is set whenever output buffers become available and is reset when none are available. That is, as long as a subport's CENSUS is less than its MAXCENSUS, meaning that at least one more record can be written, the subport's OUTPUTEVENT attribute is set.

In a COBOL WAIT statement, WRITE-OK is equivalent to ATTRIBUTE OUTPUTEVENT.

SUBFILEERROR

This attribute is set to one of the following values after each I/O, OPEN, or CLOSE operation that affects the subport file:

NOERROR
DISCONNECTED
DATALOST
NOBUFFER
NOFILEFOUND
UNREACHABLEHOST

For further information, refer to *BNA Architectual Description Reference Manual, Volume 1.*

Note: *Subport messages are queued by the MCP in storage queue (STOQ) memory blocks. When all existing storage queue blocks are full and no memory is available for allocating new blocks, the MCP displays the message NO MEM FOR STOQ and suspends any program attempting to write to a subport or to fill into a storage queue. The program is not reinstated until memory becomes available, and it is NOT notified. In particular, the SUBFILEERROR attribute value NOBUFFER is not used to indicate a lack of memory for storage queue blocks; NOBUFFER is used only to indicate that a subport's CENSUS has reached MAXCENSUS.*

TITLE

This attribute is the name of the file. It can have a length of 1 through 17 characters and must not be null. The title must match the title of the complementary file. The default is the value of the INTNAME attribute.

YOURNAME

The YOURNAME attribute of a subport is the same name as the MYNAME of the port with which it is communicating.

YOURUSERCODE

YOURUSERCODE has meaning for a subport whose SECURITYTYPE is PRIVATE. In order to connect with the PRIVATE subport, this attribute specifies the usercode under which a cooperator must be running. The default value for this attribute is the usercode of the task opening the subport. Setting the value to null sets the attribute back to the default.

Attribute's Effect on System Performance

Two attributes that can affect system performance are COMPRESSION and MAXCENSUS. The COMPRESSION attribute can be set and reset at any time after OPEN. It enables compression of the data sent between ports. Doing so is most beneficial when you are communicating between hosts over data communication lines.

MAXCENSUS sets the maximum number of messages that can be queued for input to a port or subport. When that number is reached, further attempts by the cooperator to send messages to that port or subport are rejected with a SUBFILEERROR state of NOBUFFER.

Section 8
File Handling

This section describes the way that V Series COBOL ANSI-74 handles files.

General

The approach taken in defining file information is to distinguish between the physical aspects of the file and the conceptual characteristics of the data within the file.

Physical Aspects of a File

The physical aspects of a file describe the data as it appears on the input or output media. These statements refer to the way in which logical records are grouped within the physical limitations of the file media and the means by which the file can be identified.

File Concepts

The conceptual characteristics of a file are the explicit definitions of each logical entity within the file. In a COBOL program, the input or output statements for a file refer to one logical record.

Distinguishing between a physical record and a logical record is important. A COBOL *logical record* is a group of related units of information, uniquely identifiable and treated as a unit. A *physical record* is a physical unit of information whose size and recording mode are convenient to a particular computer for storing data on an input or output device. The size of a physical record is hardware-dependent and bears no direct relationship to the size of the file contained on a device.

A single physical unit can contain one or more logical records. Or, in the case of mass storage files, a logical record can require more than one physical unit to contain it.

In this document, a reference to *records* means *logical records* unless the term *physical record* is specifically used.

The concept of a logical record is not restricted to file data, but is carried over into the definition of Working-Storage. This Working-Storage can be grouped into logical records and defined by a series of record description entries.

File Organization and Access Methods

Magnetic tape, punched paper tape, and punched card files are normally organized sequentially, and the Procedure Division of COBOL reflects this organization. Mass storage media can be used to store sequentially organized files, and the technique for doing so has been provided.

The relative and indexed input-output features provide random and sequential access to the records in a mass storage (disk or diskpack) file and prevent access to unused or deleted record positions.

A set of access routines is automatically bound into a program that declares any indexed or relative files.

Indexed and relative file formats differ from those of other files. Once created with indexed or relative organization, a file must always be declared to have that organization.

Sequential Organization

Records in a file whose organization is sequential can be accessed in the sequence established when records are written to the file. A sequential mass storage file can be used for input and output at the same time. One file maintenance method made possible by this facility is to read a record, process it, and, if it is updated, return it, modified, to its previous position.

The sequential I/O module provides the capability to access records of a file in established sequence. The sequence is established as a result of writing the records to the file. It also provides for sharing memory areas among files.

Sequential files are organized so that each record except the first has a unique predecessor record in the file and each record except the last has a unique successor record. These predecessor-successor relationships are established by the order of WRITE statements when you create the file. Once established, the predecessor-successor relationships do not change except where records are added at the end of the file.

The File-Control paragraph in the Input-Output section of the Environment Division specifies the sequential organization (ORGANIZATION IS SEQUENTIAL clause), the sequence in which records are accessed (ACCESS MODE clause, SEQUENTIAL or RANDOM), and the I/O status of the file (FILE STATUS clause.)

Relative Organization

Relative files are declared as such in the ORGANIZATION clause of the File-Control paragraph of the Input-Output section. Once a file is created with relative organization, it must always be declared to have that organization.

Relative files are accessed through the relative file intrinsic, which provides access to records in a mass storage device (disk or diskpack) file and prevents access to unused or deleted record positions. The intrinsic consists of special access routines that are automatically bound into any program that declares one or more files whose organization is relative.

The format of a relative file differs from that of other files. A relative file consists of records that are identified by relative record numbers.

The file can be thought of as a serial string of record slots, each capable of holding a logical record. Each of these record slots is identified by an integer value (relative record number) greater than zero, which specifies its logical ordinal position in the file. Records are stored and retrieved based on this number.

For example, record ten is the record associated with relative record number 10. It is stored in the tenth record slot, whether or not records have been written in the first through ninth record slots. (Conceptually, relative organization is similar to random, but relative provides protection against the accessing of unused or deleted record positions.)

In the sequential access mode, records are accessed in ascending order by relative record number. Those record slots not containing records are skipped over.

In the random access mode, the order in which records are accessed is controlled by the programmer. The desired record is accessed by placing its relative record number in the relative key data item.

In the dynamic access mode, the programmer can read either sequentially or randomly by using the appropriate form of READ statement. All other relative operations are performed randomly. The order can be sequential, but the random algorithms are used to perform the operation.

Considerations for Implementing Relative Files

Relative files differ from conventional files in the V Series systems in that they have an additional block called a *control block* and have *block control information (BCI)* appended to each data block. This additional information is interpreted by the access routines.

The following are considerations for relative files:

- A relative file can be meaningfully accessed
  - if it is declared to be of relative organization, in the File-Control paragraph of the Input-Output section, or
  - by SRTUTL, which can generate a new sequential file
- The presence of all the access routines increases the program size by approximately 12,000 bytes (excluding the buffer and the record area). Only those routines used by the program are bound into the code file, however, and they are overlayable within the user's overlay scheme.
- The block size is larger than the simple product of the record size and the blocking factor because of the presence of the block control information (BCI) at the end of each block. The BCI is computed by first multiplying the user's work area size by the blocking factor and padding the result to a word boundary. Then the BCI length (7 bytes + BLOCKSIZE) is added to this value and the final result is arrived at by padding to a word boundary once again.
- The block size for relative files is limited to 20,000 bytes.
- The relative intrinsic supports variable length records.
- The first block in the file is a control block that contains additional information about the file beyond that stored in the file header. This block is created with the file, accessed when the file is opened, and updated when the file is closed.
- The size of the control block is 56 bytes. Because it must fit into a single data block, this value is the minimum block size the relative file can have. If the block size would otherwise be less, the compiler pads it out to this size and issues a warning. If this warning is encountered, space is being wasted in each data block. Either the record size or the blocking factor should be increased in order to avoid this situation.
- The presence of the control block means that the maximum number of records that can be stored in the file is reduced by the size of one block. If this difference is significant, you should consider it when doing file size calculations.
- A relative file data block (RF block) is present in the compiled program for each relative file declared in the program. This block is 90 bytes long and contains information needed by the access routines to access and maintain the file.
- Only one buffer is allocated for each relative file. It is controlled solely by the access routines. Its size is the same as the block size computed above, plus up to 204 bytes, which are used for recovery purposes.

- A newly created (opened output) relative file is flagged as a DMSII file so that it can be recovered in the event of a failure. This then causes the MCP EOF pointer always to point to the last block in the last allocated area.
- Utility programs (such as DMPALL) used to list the file will list all the undefined blocks between the logical file EOF and the end of the area because of the difference between the control block EOF pointer and the MCP EOF pointer. This listing of undefined blocks can be prevented by directing the programs to list only the number of blocks actually contained in the file.
- The relative file being accessed might have an incorrect EOF pointer in the control block if a program or system failure occurs during an update run (open I/O). This error causes the next program that opens the file to call the automatic file recovery package and the problem is rectified.
- Simultaneous use of a relative file by an update program (one that opens the file I/O) and any other program(s) is not currently supported.
- The relative intrinsic supports a simulated FORCEIO capability to force the I/O. You invoke it by setting and resetting the FORCEIO file attribute.

  When the FORCEIO file attribute is reset (default), each block is physically written out to the file only when it becomes necessary to read in a different block, or to close the file when the block in memory has been modified. In other words, the modified block is not necessarily written out to the file after each relative update operation. Also, a relative READ causes a physical read only if the desired block is not already in the buffer.

  When the FORCEIO file attribute is set, the modified block is written out to the file at the end of each relative update operation (WRITE, REWRITE, DELETE). Also, each relative READ always performs a physical I/O on the file to ensure that the copy of the block in its buffer is the same as the copy in the file. Therefore, if the block containing the requested record is found in its buffer, the relative READ performs one of the following operations: It either reads in a fresh copy from the file if the block has not been changed, or it writes the block out to the file if the block has been changed.

  Note: *Using FORCEIO causes a significant degradation in the performance of sequential operations. If the FORCEIO capability is set for a relative file dynamically, a changed block from a prior relative update operation might not get flushed out to the file until the next relative update operation or close procedure.*

Indexed Organization

Indexed files are declared as such in the ORGANIZATION phrase of the File-Control paragraph of the Input-Output section. Once a file is created with indexed organization, it must always be declared with that organization. An indexed file is accessed in one of two ways:

- one program at a time through the indexed sequential access mode (ISAM) intrinsic, or
- more than one program at a time through the shared ISAM handler.

At compile time, ISAM access code is bound into any program that declares one or more files whose organization is indexed. ISAM provides access to the current records of the files and prevents access to unused or deleted records.

Indexed files are disk or diskpack files whose organization differs from that of other files. An indexed file consists of records, each uniquely identified by the value of one or more keys within that record. The records can be accessed in either a sequential manner or a random manner by the value of a chosen key of reference.

A record description includes one or more key data items, each associated with a key index. Each key index provides a logical access path to the data records according to the contents of the associated key data item within each record.

For the purposes of inserting, updating, and deleting records in a file, each record is identified solely by the value of its prime record key. The prime record key is declared in the RECORD KEY IS clause of the File-Control paragraph of the Input-Output section. This value must be unique. It cannot be changed when the record is updated.

Alternate record keys provide alternate access paths for retrieval of records from the file. Alternate keys are declared with the ALTERNATE RECORD KEY IS phrase in the File-Control paragraph of the Input-Output section. The value of an alternate record key can be nonunique if the WITH DUPLICATES phrase is specified.

In the sequential access mode, records are accessed in ascending order by the value of the reference key. The order of retrieval of records by use of a key index having duplicate record key values is the order in which the records were written into the file.

In the random access mode, the sequence in which records are accessed is controlled by the programmer. The programmer accesses the desired record by placing the value of one of its record keys into the appropriate record key data item before performing the access procedure.

In the dynamic access mode, the programmer can read either sequentially or randomly by using the appropriate form of the READ statement. All other indexed operations are performed randomly. The order can be sequential, but the random algorithms perform the operation.

Shared Indexed Files

Unisys V Series systems support both nonshared and shared access to indexed files. The main difference between the nonshared and shared modes is that those files used on a shared basis must be accessed through the shared ISAM handler. The shared ISAM handler enables multiple users to access the same indexed file in either Input/Output (I/O) or Input-only mode. Shared ISAM is selected by setting the SHARED attribute of the file to TRUE.

The shared implementation of shared indexed files is not intended to replace the nonshared implementation. The indexed file data files and key files can be accessed in either shared or nonshared mode; there are no differences in the files themselves. The access routines used by the shared ISAM handler for shared indexed files are basically the same as those for nonshared indexed files. The differences between the two modes is that a compiled program using an indexed file on a nonshared basis contains access routines, whereas a compiled program using an indexed file on a shared basis contains interface routines. The interface routines communicate with the shared ISAM handler, which contains the routines necessary to access and maintain the indexed file.

Programs using indexed files on a shared basis are less efficient than their nonshared counterparts because of the additional overhead involved in the communication and transfer of data between the users and the shared ISAM handler. User programs that open an indexed file for I/O access on a shared basis incur the extra overhead of implicit record locking. For this reason, user programs should not open a file for I/O access unless updates to records in the file are intended. Also, since the file cannot be shared while it is being created, there is a performance advantage to declaring the file as nonshared during file creation.

The shared ISAM handler can accommodate up to 50 users, but the limit on the number of files opened by all users at any given time is 25. If the limitations of using an indexed file in shared mode prove too restrictive, Unisys recommends upgrading to Data Management System II (DMSII).

Programs accessing indexed files in nonshared mode can control the number of key buffers allocated to each file. Programs that access indexed files on a shared basis cannot specify the number of key buffers per file. That action is controlled by the shared ISAM handler.

Note: *At various times during the execution of a user program that accesses one or more shared indexed files, use a* ?<mix #> WY *command to show that the job is WTG DMS FUNCTION. This message indicates that the user's job has issued a request to the shared ISAM handler and is awaiting a response.*

Considerations for Implementing Indexed Files

Indexed files differ from conventional files on the V Series systems in that they have a block of data called a *control block* at the beginning of the file and additional block control information (BCI) appended to the end of each block. In order to interpret this additional information, compiler-provided intrinsics are bound into all programs that declare one or more indexed files.

An indexed file can be either nonshared (that is, to be used by one user who has opened the file for I/O access at one given time) or shared by multiple users. A user-program operation on a nonshared indexed file generates a call to the appropriate intrinsic access routine, and that routine accesses the files. When the indexed file is used on a shared basis, interface routines, which are bound into the program during compilation, allow the program to communicate with the shared ISAM handler. The shared ISAM handler then carries out the access requests.

Considerations for Implementing Nonshared and Shared Files

The following considerations are applicable for an indexed file used on either a nonshared or shared basis:

- An indexed file declared in a program is really a group of files: a data file and one or more key files. Outside the code file, these files are not recognized as related. You must specify all files in dumps, loads, and so forth.
- The naming convention for key files is the following: The prime key file name consists of the first four characters of the data file name with 01 added at the end (e.g., if the data file were ISAMFL, the prime key file name is ISAM01). The name of the first alternate key file consists of the first four characters of the data file name with 02 added (ISAM02). The names of other alternate key files continues similarly (ISAM03, ISAM04, etc.).
- Each key declared for each indexed file causes a key file to be created to contain the resulting key index. A file information block (FIB) and associated control information is allocated in the program for each key file. The net effect is to increase program size by 220 bytes per indexed file key.
- The data block size is larger than the simple product of the record size and the blocking factor because the Block Control Information (BCI) is at the end of each block. Compute the total block size by first multiplying the user work area size by the blocking factor and padding the result to a word boundary. Then add the BCI length (7 bytes + BLOCKSIZE) to this value and then pad to a word boundary once again to arrive at the final result.
- The first block of the file is a control block, which contains additional information about the file beyond that stored in the file header. This block is created when you create the file, accessed when you open the file, and updated when you close the file.

- The size of the control block is 126 bytes plus 50 bytes for each alternate key declared for the file. Because it must fit into a single data block, this block size is the minimum the data file can have. If the block size would otherwise be less, the compiler pads it up to this size and issues a warning. If this warning is encountered, the file is not recoverable in the event of a failure and space is being wasted in each data block. In order to avoid these two problems, either the record size or the blocking factor must be increased sufficiently to make the data block size larger than the control block size.
- The presence of the control block means that the maximum number of records that can be stored in the file is reduced by the size of one block. If this difference is significant, you should consider it when doing file size calculations.
- An indexed file data block (IF block) is present in the compiled program for each indexed file declared in the program. This block contains certain information needed by the access or interface routines to access and maintain the file. Its size is 190 bytes plus 50 bytes for each alternate key declared for the file.
- A block (data file block or key file block) is physically written only when it is necessary to read in a new block or to close the file and when the current block in memory has been modified. Blocks are not necessarily written after every record update. The one exception occurs when the FORCEIO file attribute is true. See "Implementation Considerations for Nonshared Files" in this section.
- A newly created (opened output) indexed data file is a DMSII file so that it can be recovered in the event of a failure. Doing so causes the MCP EOF pointer always to point to the last block in the last allocated area.
- Utility programs (such as DMPALL) used to list the file will list all the undefined blocks between the logical file EOF and the end of the area because of the difference between the control block EOF pointer and the MCP EOF pointer. This listing of undefined blocks can be prevented by directing the programs to list only the number of blocks actually contained in the file.
- If a program or system failure occurs during a file update, the indexed file being accessed probably has unmatching data and key files and might have an incorrect EOF pointer in the control block. The in-use flag in the control block then remains set to indicate the problem and a 9:6 error indication (see "Indexed File Error Indications," in this section) will be returned if an open statement is attempted.
- An indexed file recovery is performed by opening the old data file input with relative organization. The data records must then be read sequentially, using the relative intrinsic and written out to a new indexed file, using the indexed intrinsic. A recovery program source template that does this task is provided to assist users in recovering their files.
- The maximum size limit for indexed file keys is 96 characters.
- A maximum of 99 keys can be declared for an indexed file.
- The indexed intrinsic supports variable length records.
- The block size for indexed sequential files is limited to 20,000 bytes.

- Nonshared indexed file jobs cannot access an indexed file if the shared ISAM handler has opened the same file for multiuser access. Also, the shared ISAM handler cannot access an indexed file if a user has opened that file to be used on a nonshared basis for updating.
- There are certain factors that you should attend to in order to ensure optimal performance on indexed files. The first is the number of keys declared for a file. Each file update requires that all the key files be updated (except REWRITES that don't change the keys); the number of keys, therefore, is directly proportional to the time required to perform an update. Obviously, this consideration is not important if the file is infrequently updated.

  Next is the data file blocking factor, which is specified in the file declaration. You can achieve optimal random performance (other than WRITEs during a file load) by using a small blocking factor. You can achieve optimal sequential performance (and WRITEs during a file load) by using a large blocking factor. If both are necessary, then an intermediate blocking factor determined by trial and error is called for.

  The larger the key file blocking factor is for a given key, the faster is the access and update of that key file. You have only indirect control over the key blocking via choice of key length and key type. An indexed file key block is targeted to be approximately 2250 bytes, based upon the largest integer number of key entries that are less than that size. Therefore, the smaller the key size, the larger the blocking factor. You can minimize key size by minimizing the key length.

  Applying updates in key value order of one of the keys can reduce the number of I/Os on that key file significantly. The closer the key values are in each successive update record, the higher the probability that the same key tables will be used and the greater the I/O savings. Similar savings can occur if the updates are not precisely ordered, but tend to cluster tightly by the value of one of the keys. In both of these cases, the key of reference should be set to the ordered key prior to the updates because the buffer algorithm gives priority to the key tables of the key of reference.

  When an indexed file is used in a nonshared mode, allocating sufficient key buffers can reduce the overall number of key file I/Os significantly. The default is three due to historical precedent, but this number is usually not enough and can cause a performance bottleneck. See "Indexed File Buffers" in this section for specific recommendations.

Implementation Considerations for Nonshared Files

The following considerations apply only to an indexed file used on a nonshared basis (SHARED file attribute is FALSE):

- An indexed file can be meaningfully accessed
  - if it is declared to be of indexed organization in the File-Control paragraph of the Input-Output section, or
  - by SRTUTL, which can generate a new sequential file

- The presence of all the access routines increases the program size by approximately 31,000 bytes (excluding buffers, FIBs, BCS and record area). Only those routines used by the program are bound into the code file, however, and they are overlayable within the user's overlay scheme.
- Each of the buffers allocated for an indexed file has a descriptor element associated with it. These elements make up the buffer control structure (BCS) which is pointed to by the BCS pointer in the indexed file data block (IF block) (see "Considerations for Implementing Nonshared and Shared Files" earlier in this section).
- Only one buffer is allocated for the data file of each indexed file. Its size is the same as the data block size plus up to an additional 204 bytes which are used for recovery purposes.
- The user can specify the number of key buffers allocated for each indexed file (3 through 98; default=3). They are approximately 2250 bytes each in size. These buffers are shared among all the key files associated with that indexed file.
- The in-use flag in the data file control block is set so that, when an indexed file is opened I/O by a user program, the file cannot be opened again by another user program until it is successfully closed by the first user program.
- The indexed intrinsic supports a simulated FORCEIO capability. It is invoked by setting and resetting the FORCEIO file attribute.

  When the FORCEIO attribute is reset (default), each block (data or key block) is physically written out to the file only when it is necessary to read in a different block, or to close the file when the block in memory has been modified. In other words, the modified blocks are not necessarily written out to the file after each indexed update operation. Also, an indexed READ procedure causes a physical read only if the desired block is not already in the buffer.

  When the FORCEIO attribute is set, all modified data and key blocks are written out to their respective files at the end of each indexed update operation (WRITE, REWRITE, DELETE). The order they are written in is undefined. Also, each indexed READ operation always performs physical I/Os on the data and key files to ensure that each copy of the desired data and key blocks in its buffers is the same as the corresponding copy in the files. Therefore, if the data or key block containing the desired record or key entry is found in one of its buffers, the indexed READ operation does one of the following: it reads in a fresh copy from the file if the block has not been changed, or it writes the block out to the file if it has been changed.

  Note: *Using FORCEIO causes a significant degradation in performance. It thwarts the activity of the key buffer algorithms and renders additionally allocated key buffers useless. FORCEIO defeats most of the performance advantages inherent in sequential operations. If FORCEIO is set for an indexed file dynamically, one or more changed data or key blocks from a prior indexed update operation might not get flushed out to the file until the next indexed update operation or close procedure.*

- Programs using indexed files on a nonshared basis can declare alternate keys in any order. Programs using indexed files on a shared basis must declare alternate record keys in the same order as that in which the keys were declared in the program that created the file.

Implementation Considerations for Shared Files

When an indexed file is used on a shared basis, the following considerations apply:

- The file is accessed only if it is declared to have indexed organization in the File-Control paragraph of the Input-Output section.
- The minimum hardware required for operation of the shared ISAM handler is a V Series processor with main memory of 300–999KD in excess of the MCP requirement and an additional amount of memory sufficient to hold the user programs that contain the shared ISAM interface routines.
- The necessary software environment requires a Master Control Program (MCP) to be resident in main memory. The MCP release level must be compatible with the release level of the shared ISAM handler. The DMSII extension module must be available on the system, and the DMS option of the MCP must be set.

  In addition to the preceding requirements, the following two files must be available on disk:

  - ISAMDB. The shared ISAM handler object code file.
  - ISHxxx. The ICM file containing a set of intrinsics that interface with the handler and are bound into user programs by the appropriate compiler. (The xxx is a number reflecting the release level of the intrinsics. The ICM file for MCP/VS 2.0 would be named ISH200.)
- The ability to share an indexed file through the shared ISAM handler is only available after the file has been created. An indexed file cannot be shared by other users during file creation.
- The shared ISAM handler can accommodate up to 50 users, but the maximum number of users active at any one time is controlled by the LIMIT DMS USERS record in the system configuration file. The shared ISAM handler does not limit the number of files any single user can access, but only 25 files can be opened at any given time.
- Each user program accessing a shared ISAM file must declare both the file and the corresponding record keys and record descriptions in exactly the same manner as they were declared when the file was created. For example, if the program creating the file declares the file as having 5 record keys, then all programs accessing this file as a shared file must also declare it as having 5 record keys. Furthermore, the record key descriptions as well as the order in which the record keys appear must be the same as those described in the program creating that file.

- The file attributes of a shared indexed file cannot be changed after the shared ISAM handler has opened the file. Its attributes are those specified by the first user program that logically opened the file. The word *logically* means that the user program does not actually open the file; the shared ISAM handler does. The MCP does not have any information linking the shared indexed file to the user program. Consequently, any user program interrogating the file attributes of a shared indexed file might or might not receive the actual file attribute values of the file (i.e., the values specified when the file was opened). Whether or not the attribute values are the correct ones depends on the particular attribute (some of which are always returned correctly) and on the sequence of user programs that have opened the file (the first user program to open the file always receives the correct attributes).
- The number of key buffers allocated to a shared indexed file is controlled by the shared ISAM handler. Also, both the BCS and the IF Block are in the shared ISAM handler. You cannot determine the number of key file buffers since user specifications are ignored.
- Security consideration for shared indexed files are as follows:
  - In order to be shared, an indexed file must be created with a SECURITYTYPE file attribute of either PUBLIC or GUARDED.
  - If the SECURITYTYPE is GUARDED, the shared ISAM handler (ISAMDB) must be included in the guard file as an authorized user.
  - In order to be shared, an indexed file must have a SECURITYUSE file attribute of IO.
  - Indexed files created in the shared mode are created with a SECURITYTYPE of PUBLIC and a SECURITYUSE of IO.
- A major problem in a shared file environment that allows updates by multiple users is protecting concurrent updates to different fields of the same record. Consider the case of 2 programs accessing a shared indexed file for I/O access in a sequential access mode. The following chain of events could occur:
  1. User A reads record 1 -- Record 1 = [X, Y]
  2. User B reads record 1 -- Record 1 = [X, Y]
  3. User A updates X to M and rewrites record 1 -- Record 1 = [M, Y]
  4. User B updates Y to N and rewrites record 1 -- Record 1 = [X, N]

     The final incorrect content of record 1 would be [X, N]. This error would occur because the update at step 3 was lost. The correct content of record 1 should have been [M, N]. This problem occurs because both users were allowed to have a copy of the record and each was allowed to modify the record.

- To prevent this problem from happening, the shared ISAM handler uses an implicit record locking mechanism to protect concurrent record updates. This mechanism locks a record that is to be updated. Any attempt to read a locked record is suspended until the record is released. The following example illustrates how record locking changes the preceding sequence of events. (Remember that this example assumes that both users open the file for I/O access. If user B, for example, opens the file for READ only access, the action in step 2 would be successful rather than suspended.)

  1. User A reads record 1 -- Record 1 = [X, Y], Record 1 is locked
  2. User B reads record 1 -- User B is suspended
  3. User A updates X to M and rewrites record 1 -- Record 1 = [M, Y], record 1 is released
  4. User B is reinstated and reads record 1 -- Record 1 = [M, Y], record 1 is locked
  5. User B updates Y to N and rewrites record 1 -- Record 1 = [M, N], record 1 is released

  Every READ action by a user who opens the file for I/O access implicitly locks the record. The next access of the file by the same user releases the record.

  Note: *Implicit record locking does not always guarantee that all modifications to the record will be reflected. For example, in a random access mode, a user can rewrite a record without reading it up first. In this case, the random rewrite action can destroy previous updates. This situation can be avoided if users of shared files always precede update actions with read operations and modify only the appropriate field or fields.*

- The implicit record-locking feature can possibly introduce deadlocks. Consider the following chain of events.

  1. User A reads file X, record 1 -- File X, record 1 is locked
  2. User B reads file Y, record 2 -- File Y, record 2 is locked
  3. User A reads file Y, record 2 -- User A is suspended
  4. User B reads file X, record 1 -- User B is suspended

  In this example, both users are suspended at the conclusion of step 4, each waiting for the other to release a locked record. To resolve such situations, the shared ISAM handler detects deadlocks and randomly terminates one of the user programs involved, allowing the other users to proceed.

- The shared ISAM handler initially occupies 362 KD of memory and can grow. The maximum size is limited to 999 KD. The growth of the shared ISAM handler occurs when more shared indexed files are opened and the initially allocated memory is not enough to accommodate such things as the various FIBs and buffers. The extra memory requirements for the shared ISAM handler are justified by the reduction in code file size of the user's programs. Memory requirements for these programs are reduced since the various data and key buffers are no longer necessary in each program.

Indexed File Buffers

The following describes buffer considerations for indexed files.

Nonshared vs. Shared Indexed File Buffer Considerations

Programs that use indexed files on a nonshared basis can control the number of key buffers allocated to each file. However, programs that use indexed files on a shared basis cannot specify the number of key buffers; the shared ISAM handler controls buffer decisions.

Considerations for Nonshared Indexed File Buffers

When an indexed file is used on a nonshared basis, the host compiler allocates from 4 to 99 buffers for each indexed file. The user program cannot directly access these buffers at any time and other files cannot share them.

The first buffer of each indexed file is used exclusively for data file I/Os. The other 3 to 98 are shared by the key files for their I/Os. The blocking factor and the BCI size determine the size of the data buffers. The key buffers size is approximately 4500 digits. The block size for each key file is as close to this value as possible, and the key buffer size is equal to the largest key file's block size.

The user interface for specifying the number of indexed file key buffers allocated is by the use of the RESERVE nn AREAS clause in the SELECT statement in the File-Control paragraph of the Input-Output section. In the RESERVE nn AREAS clause, *nn* means allocate 1 buffer for the data file and (nn−1) buffers for the key files. The value of *nn* can range from 4 (specifying 3 key buffers) through 99 (specifying 98 key buffers). If the value is not specified, the default is three key buffers.

When determining how many key buffers to allocate for an indexed file, you should consider several factors. The type of operations you are performing affects the number needed. Update operations generally involve accessing and updating all the key files, so they need more buffers than inquiry operations, which usually involve reading only one key file. Therefore, the more keys the file has, the more key buffers are recommended when you are updating. Sequential updates should have more buffers than random updates. The order in which the updates occur also influences the number of key buffers needed. The number of levels in each of the key indexes is also important.

Based on these and other considerations, Unisys has compiled a list of minimum key buffer recommendations which are shown in Table 8–1. The intrinsic operates correctly in all cases with its lower limit of 3 buffers, but allocating fewer than the recommended number usually leads to thrashing. You should start with the recommended number of buffers and then add more for certain types of operations until the tradeoff between improved key file I/O performance and memory size becomes unacceptable (each key buffer is approximately 2250 bytes).

Table 8–1. Minimum Key Buffer Sizes

Access Method	Minimum Recommended Key Buffer Size
One Key (Prime Key) Only	
Sequential inquiry	3
Random inquiry	Number of index levels
Sequential update	Number of index levels + 1
Random update	Number of index levels + 1
Multiple Keys	
Sequential inquiry	3
Random inquiry	Number of index levels of tallest inquired key + number of keys inquired upon − 1
Sequential update	Number of keys + number of index levels of tallest key + number of levels of key of reference − 1
Random update (totally random)	Number of keys + number of index levels of tallest key

In general, sequential inquiry operations do not show any improvement when you add additional buffers beyond the recommended number:

- Sequential update with one key improves very little.
- Sequential update with multiple keys improves similarly to random update because one or more of the keys is probably random with respect to the update order.
- Random inquiry improves until the number of buffers is sufficient to hold all of the key tables for the key or keys being referenced.
- Random update also improves until the number of buffers is sufficient to hold all of the key tables for the keys being updated.

Additional Buffer Considerations

The following are additional buffer considerations that apply to nonshared index files.

- REWRITES with multiple keys that change most or all of the keys tends to need more than the recommended number of buffers.
- REWRITES with multiple keys that change none or very few of the keys do not need all the recommended buffers.
- Additional buffers are recommended when a file containing many duplicate key values is updated.
- Whenever random updates are applied in order of one or more keys, or if the order tends to cluster according to one or more key orders, additional buffers are recommended for additional performance gains.
- Programs that load indexed files or perform large numbers of indexed write operations show dramatic improvements as the number of key buffers increases, so they should have as many as possible. The one exception case is a single key file in which the order of the write operations is sequential.

Indexed File Recovery

If the FORCEIO file attribute is reset and a program has a nonshared indexed file open I/O that fails to close, the file must be recovered. The EOF pointers in the control block are only updated when the file is closed and might be incorrect. Any partially completed update would leave the key files corrupted and/or out of sync with the data file.

If the FORCEIO file attribute is set when a program that has an indexed file open I/O fails to close, and that file is being used on a nonshared basis, the file needs to be recovered only if the program was doing an update function (WRITE, REWRITE, DELETE) when it failed. At the end of each update function, all modified key file blocks and the data file control block are rewritten.

The failure to close the file causes the in-use flag to remain set. All programs attempting to open the file then check the file header to see if it is in use by an update program. If the file really is in use, the programs branch to their user-declared exception routines and the system assigns the associated status digits the value 9:7. If no exception routine is declared, the program is terminated. If the file is not in use by an update program, the programs then check the update process flag to see if the file was being modified when the failure occured. If the flag is set, they will branch to their user declared exception routines and the associated status digits are assigned the value 9:6. If no exception routine is declared, the program is terminated.

When you are using an indexed file on a shared basis, the shared ISAM handler closes the file, if necessary. However, the shared ISAM handler does not close the file if the file becomes corrupted. Then it must be recovered.

An indexed file is recovered by reading up the file records from the data file of the corrupted indexed file, using the relative intrinsic, and writing the records out to a new indexed file, using the indexed intrinsic.

Note: *The relative intrinsic might not be able to access all of the records present in the corrupted indexed file unless it is opened input and the access type is sequential.*

At worst, the data block being modified in memory at the time of the failure will be lost and the changes to it will not be reflected in the recovered file.

A recovery program source template is provided to assist users in recovering their indexed files. You must fill in the appropriate file declarations, compile the COBOL source, and then execute the compiled program to recover the corrupted indexed file. Unisys strongly recommends that users replace any existing recovery programs with appropriately modified versions of this tool.

The implementation of indexed file recovery requires that the MCP be told that indexed files are DMSII files. Doing so causes the MCP EOF pointer always to point to the end of the last allocated area. This causes certain operational changes which are discussed earlier in this section under "Considerations for Implementing Nonshared and Shared Files." The MCP is not told that indexed files that are opened input only are DMSII files.

To ensure that your indexed files are recoverable, you should check your compile listings for the warning "BLOCKSIZE PADDED - FILE NOT RECOVERABLE". If you encounter this message, you must increase the blocksize to at least the control block size. See "Considerations for Implementing Nonshared and Shared Files" in this section.

Indexed File Error Indications

The FILE STATUS data item notifies a program that an error or some other noteworthy situation has occurred on an indexed file. This data item can be declared in the Data Division as a two-character alphanumeric data item and in the FILE STATUS phrase of the File-Control paragraph of the Input-Output section. If the user has declared a FILE STATUS data item for the file, the ISAM intrinsic assigns a two-character value to completion of each operation. The two characters are known as *status digits* and indicate the outcome of the operation.

Nonserious status digit indications (values 0:0 thru 0:2) can be checked for immediately after the operation; serious indications (values 1:0 thru 9:7) can be checked in exception routines; fatal indications (values F:0 thru F:A) are displayed in an error message just prior to termination. Table 8–2 presents a list of the values that the status digits can be assigned and the meanings of those values. These status key explanations apply to indexed files used on either a nonshared or shared basis.

Table 8–2. Indexed File Status Conditions

Status Digits	Exception Explanation
0 0	Function was successful (all)
0 2	Next record has duplicate key (READ) Duplicate key entry found during key file insertion (WRITE, REWRITE, DELETE recovery)
1 0	Logical EOF exceeded (sequential READ) File empty (READ, START)
2 1	Prime key changed between last read and this function (sequential REWRITE)
2 2	Duplicate key entry found, but not allowed on this key (WRITE, REWRITE, DELETE recovery)
2 3	No record found in file with this key (random READ, START=, REWRITE, DELETE) No record found in file with key > or >= this key (START>, START>=) Logical EOF exceeded (> EOF, = 0) (START, REWRITE, DELETE)
2 4	Data file physical EOF exceeded (WRITE) Key file physical EOF exceeded during table split (WRITE, REWRITE, DELETE recovery)
9 1	Open type incompatible with requested function: -- READ with file not OPEN or OPEN OUTPUT -- START with file not OPEN or OPEN OUTPUT -- WRITE with file not OPEN or OPEN INPUT -- REWRITE with file not OPEN or OPEN INPUT or OUTPUT -- DELETE with file not OPEN or OPEN INPUT or OUTPUT
9 2	Current record not defined (sequential READ)
9 3	Current key of reference undefined (= 0, > number of keys) (READ, START)
9 4	Last function was not a successful sequential read (sequential REWRITE, sequential DELETE)
9 5	Variable length records: new record length differs from old; operation aborted (REWRITE)
9 6	File recovery required due to termination of previous update program while an update function was in progress (OPEN INPUT, IO)
9 7	File in use by updating program (OPEN INPUT, IO)
F 0	Unexpected access routine EOF (block > MAX block, = 0) on data or key file (READ, START, WRITE, REWRITE, DELETE) IF block smashed: IF block FD didn't match FD in the control block (CLOSE)

continued

Table 8–2. Indexed File Status Conditions (cont.)

Status Digits	Exception Explanation
F 2	File corrupted Search key > highest table entry (READ, random REWRITE, random DELETE) Key file entry pointed to empty data record slot (READ, random REWRITE, random DELETE)
F 3	File corrupted: update function in process failed and could not be recovered (WRITE, REWRITE, DELETE)
F 4	Data file corrupted: data block in free space list had no empty record slots (WRITE)
F 5	Mismatched file versions: key file date/time stamp did not match data file date/time stamp (OPEN)
F 6	Data or prime key file attribute mismatch: IF block FD did not match control block FD (OPEN INPUT, IO)
F 7	Missing or mismatched alternate key declaration Number of keys in IF block differs from control block (OPEN IO) IF block FD unmatched in control block (OPEN INPUT, IO)
F 8	No longer used
F 9	CLOSE attempted on a file that was not OPEN OPEN attempted on a file that was already OPEN
F A	Unexpected buffer locking deadlock (WRITE, REWRITE, DELETE recovery)

When the intrinsic encounters a fatal error, the ODT displays the appropriate fatal error message, and the program terminates abnormally (invalid instruction EC). The error message contains an indication of the problem and the value of the status digits. Serious errors (values 1:0 thru 9:7) are also fatal if the user has not declared an exception routine. A list of the possible fatal error messages applicable to indexed files used on either a nonshared or shared basis is presented in Table 8–3.

Table 8–3. Indexed File Fatal Error Messages

Status Digits	Error Message Text	Message Code
F 0	<file name> : READ/WRITE EOF ERROR	(ERR=F0)
F 1	<file name> : IF/RF BLOCK SMASHED	(ERR=F1)
F 2	<file name> : INDEXED FILE CORRUPTED	(ERR=F2)
F 3	<file name> : UPDATE RECOVERY FAILURES	(ERR=F3)
F 4	<file name> : FREE BLOCK LIST ERROR	(ERR=F4)

continued

Table 8–3. Indexed File Fatal Error Messages (cont.)

Status Digits	Error Message Text	Message Code
F 5	<file name> : DATE/TIME STAMP MISMATCH	(ERR=F5)
F 6	<file name> : FILE ATTRIBUTE MISMATCH	(ERR=F6)
F 7	<file name> : MISSING KEY DECLARATION	(ERR=F7)
F 8	No longer used	
F 9	<file name> : ALREADY OPENED/CLOSED	(ERR=F9)
F A	<file name> : BUFFER HANDLING DEADLOCK	(ERR=FA)
x x	<file name> : NO USER EXCEPTION ADDRESS	(ERR=xx)

Programs that open indexed files for shared access can encounter error conditions applicable to all indexed files, shared or nonshared. In addition to these indexed file errors, there are error situations that are unique to a shared indexed file environment. For example, a shared indexed file could be corrupted by one user, causing all subsequent users accessing this corrupted file to encounter an error. Shared indexed file errors are fatal errors, so the user program terminates whenever it encounters such errors. A list of errors only for shared indexed files is presented in Table 8–4.

Table 8–4. Indexed File Shared Error Messages

Status Digits	Error Message Text	Message Code
A 0	<file name> : FILE ABSENT/INV SECURITY	(ERR=A0)
A 1	<file name> : INCOMPATIBLE OPEN TYPE	(ERR=A1)
A 2	<file name> : UNKNOWN REQUEST	(ERR=A2)
A 3	<file name> : FILE CORRUPTED	(ERR=A3)
A 4	<file name> : HANDLER TERMINATED	(ERR=A4)
A 5	<file name> : XFER DATA ERROR	(ERR=A5)
A 6	<file name> : TOO MANY USERS	(ERR=A6)
A 7	<file name> : DS-ED DUE TO DEADLOCK	(ERR=A7)
A 8	<file name> : WAIT QUEUE OVERFLOW	(ERR=A8)
A 9	<file name> : LOCKED RECORD OVERFLOW	(ERR=A9)
A A	<file name> : FILE NOT OPEN	(ERR=AA)

Shared File Functions

Shared file functions permit multiple programs or processors to access a disk or diskpack file simultaneously, when the processor or processors are configured as a shared system. Fundamental to the handling of shared files is the ability of a program to lock a block of a shared file, thus denying other programs on the same or other processors access to that block until it is unlocked. Doing so prevents two programs from simultaneously updating the same record. Since locking occurs only at the block level, unlocked blocks remain accessible to other programs.

In order to execute a program that uses shared file statements, the MCP options SHRD (shared disk) and/or SHRP (shared pack) must be set.

Refer to *V Series Systems Operations Guide Volume 3: System Utilities* for further information on shared files and programming considerations.

The following paragraphs describe the COBOL syntax required to declare and to access shared files.

SELECT Clause

Files that are declared shared must be random access files, and you must use the ACTUAL KEY clause and declare a FILE STATUS variable. If a stalemate condition occurs, a value of 96 is placed into the FILE STATUS variable.

File Attribute SHARED

Declare a file as shared by setting the file attribute SHARED to TRUE, either in the FD or in the Procedure Division.

A file must be declared as shared prior to being opened. The compiler does not check to ensure that a file was declared as shared prior to its use.

The shared file functions implemented as system dependent intrinsics (SDIs) are LOCKONLY, UNLOCK, READLOCK, READLOCKWAIT, SEEKLOCKONLY, SEEKLOCK, SEEKLOCKWAIT, and WRITELOCK. The SDIs are discussed in Section 6 "Procedure Division" under "CALL INTRINSIC Statement".

In addition, you can use normal READ, WRITE, OPEN, and CLOSE statements with shared files.

DECLARATIVES Section

A USE AFTER STANDARD EXCEPTION procedure must be defined in the program's Declaratives section for shared files if ON OVERFLOW clauses are not provided in the appropriate intrinsic calls. Unisys recommends that you always declare a USE procedure for a file in case you omit any ON OVERFLOW clauses.

File-access exceptions, such as invalid key and stalemate, are dealt with in the USE procedure or in the ON OVERFLOW clause of the intrinsic call.

Exception-Handling Procedures for Shared Files

The READLOCKWAIT, READLOCK, and WRITELOCK intrinsics require either an ON OVERFLOW clause or an applicable USE procedure in the program's Declaratives section. This exception routine specifies the action to be taken if an access error (such as invalid key or stalemate) occurs. If a stalemate condition occurs, the file's FILE STATUS variable will contain the value 96.

Port Files

Port files permit programs running on the same or different processors to exchange data. The port file is a means of interprogram communication. Ports are somewhat like the older core-to-core and storage queue mechanisms (refer to the SEND/RECEIVE verbs), but differ from them in that a program can communicate with another program through a port file.

Port files are designated as local-only ports when both programs are executing on the same processor, and as remote ports when the programs are on different processors. The only difference between the two is that, to use remote ports, both systems need BNA software, but to use local-only ports, a V Series system needs only the current MCP with the STOQ option set.

To use port files, each program declares a file whose type is PORT. If remote ports are to be used, each program specifies the host name of the remote port file with which it wishes to communicate. If, when the file is opened, the corresponding file is found, a dialogue is established between the two programs. Records written to the port file of either program can be read from the port file of the corresponding program. Further selectivity is possible by declaration of subports.

The port file can be viewed as a more versatile substitute for the storage queue mechanism. The speed of local-only port communication approximates that of the storage queue mechanism. For further information see "Port Files" in *V Series MCP/VS Programming Reference Manual.*

Note: *A port file cannot be specified in an MCP FILE statement.*

Section 9
Compiler Operation

General

COBOL ANSI-74 is a 280KB compiler designed to operate with MCP/VS.

Compilation File

To compile a COBOL Source Language Program, present a file to the MCP for compilation. This file appears as in Figure 9–1. The following paragraphs define the elements of the compilation file.

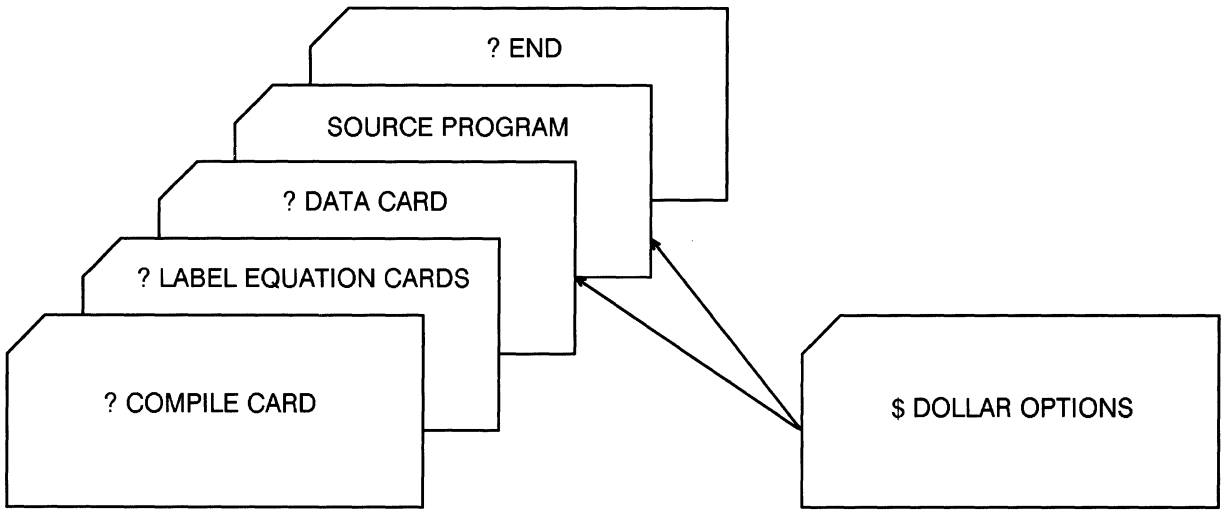


Figure 9–1. Compilation File

COMPILE Statement

The first record is a COMPILE control instruction that instructs the MCP to start the COBOL compiler and to compile the indicated program-name.

The following is the format of this statement:

```
{ CMP     }
{ COMPILE } program-name    [ ON object-pack-name  ]

    [ WITH ]      COBOL  [ ON compiler-pack-name ]   [ (0, "Lnnnn") ]
                                                     [ (,"L")       ]

    [ LIBRARY ]
    [ SYNTAX  ]       [ MEMORY  [ + ]  integer ]
    [ SAVE    ]

    [ { INSERT }       0 4 UA "abcd" ]
    [ { INSERT }       40 6  <nnn000> ]
```

If the compile control record is not in the input file, you can manually enter the COMPILE control instruction at the ODT.

The ON object-pack-name option sends the resultant program-name to the pack specified by the pack-name. Any ICM generated goes to the system default pack.

The ON compiler-pack-name option causes the MCP to search for and use the COBOL compiler, which has been copied to the specified diskpack.

Use the "L" option when you must compile a program of over 90000 source lines. This option increases the size of the compiler work file. For large programs, it might be necessary to increase the compiler's stack. Use the following clause to increase the stack size

INSERT 40 6 *nnn*000

The *nnn* construct is the new size of the stack, in thousands of digits. A recommended minimum value is 020. When the stack is increased, you must increase the compiler memory by at least the same amount, using the MEM+*nnn* command (see the following paragraphs).

The COBOL compiler uses available memory (over .5MB) for its workfiles on systems not using QWIKDISK.

The nnnn parameter is the amount of memory in thousands of digits given to the workfiles. The following is an example:

```
COMPILE MYTEST WITH COBOL (0, "1500") LIB
```

The example syntax allows the COBOL compiler to use up to 1.5MD of high memory for its workfiles.

The LIB or LIBRARY option instructs the MCP to compile the program specified by program-name and place the resultant object code into the system library.

The SYNTAX option instructs the MCP to compile the program specified by program-name for a syntax check only. If the compilation process is successful (no syntax errors are encountered), no code file or ICM is generated.

The SAVE option instructs the MCP to compile the program specified by program-name, place the resultant code into the system library, and then execute the object code.

If neither LIB, LIBRARY, SYNTAX, or SAVE is specified, the MCP compiles and then executes the program specified by program-name and does not save the code file.

The MEM option instructs the MCP to give the compiler more memory space. The integer entry can be either a 3- or a 6-digit integer. A 6-digit entry is rounded up to 3 digits, indicating memory in thousands of digits (KD). The +integer option means that the indicated memory is to be added to the minimum memory required. The absence of the + means that the integer represents the total memory requirement. In any case, the memory allocated by the MCP is never less than the size of the compiler.

The COBOL compiler, by default, expects a card file (internal file name CARD) as input. You can override this default by adding the INSERT 0 4 UA "abcd" clause to the COMPILE command.

In this clause, *a*, *b*, *c*, and *d* represent alphabetic character positions. Each position must be filled by a letter or a blank, and the string of four characters must be enclosed in quotation marks.

Each position within the 4-character string represents an input file or file format to the compiler. The character in the position indicates either the device type on which the file is located or the format of the file, as shown in Tables 9–1 and 9–2.

Note: *You must use the INSERT 0 4 UA "abcd" clause if the primary symbolic file is anything other than a card file or a disk file with eighty characters per record and one record per block.*

Table 9–1. Valid INSERT Values for Cardless Compiles

Position	Meaning	Permissible Values	Default Values
a	Primary symbolic file	C,D,P,T,*s*	C
b	Secondary symbolic file	C,D,P,T,*s*	D
c	Output symbolic file	C,D,P,T,*s*	D
d	Primary file is editor format	E, *s*	*s*

s Represents a space.

Table 9–2. Meanings of INSERT Values for Cardless Compiles

Position	Character	Meaning
a, b, c	C	Card (Reader for a,b; Punch for c)
	D	Disk (Any 100-byte)
	P	Diskpack (180-byte)
	T	Magnetic Tape
	s	Device unspecified
d	E	Editor-format primary symbolic file
	s	Primary symbolic is non-editor file

s Represents a space.

Indicate the input and output files for the compile operation by supplying the appropriate combination of permissible values from the second table in positions a, b, c, and d of the INSERT clause. Once you have specified the media and format of the input and output files, you can give file names on dollar records or in FILE-equate clauses if the actual names of the files are other than the default names assumed by the compiler.

The INSERT 40 6 <nnn000> clause specifies a large stack size for the compiled COBOL program. The six-digit number is the additional memory, in KD, to be used for stack.

Table 9–3 summarizes the input and output files and formats acceptable when you use the COBOL compiler (an X indicates an acceptable file), the compiler's internal name for each file (for use in FILE equating), and the values that must be supplied in the INSERT clause when you compile without cards.

Table 9–3. Compiler Files and Formats

		Device / Format						
Compiler I/O File	**Compiler File Name**	**Card 80×1**	**Disk 80x ***	**Disk Editor**	**Disk pack 80x ***	**Disk pack Editor**	**Tape 80x ***	**Print 132×1**
Primary Symbolic	CARD** or EDITFL**	X (a=C)	X (a=D)	X (a=D, d=E)	X (a=P)	X (a=P, d=E)	X (a=T)	
Secondary Symbolic	SOURCE	X (b=C)	X (b=D)		X (b=P)		X (b=T)	
Output Symbolic	NEWSOU	X (c=C)	X (c=D)		X (c=P)		X (c=T)	
Output Listing	LINE							X

* Source input files can be any blocking factor from 1 through 9. Any output source file is blocked 9.

** The name of the primary file is CARD when it is a non-editor-format file, and EDITFL when it is an editor file.

DISK is the default medium for the primary symbolic file when Editor format is specified.

The values specified in an INSERT clause take precedence over information in dollar records. A dollar record can supplement the information in an INSERT statement (for example, provide the name of the secondary or output symbolic file).

The following are examples of compile statements for cardless compiles:

```
COMPILE PROG1 COBOL LIB IN 0 4 UA "D   " FILE CARD = PROG1S
```

The program PROG1 is compiled. Input is a disk file named PROG1S. There is no other input file and no output symbolic file is created.

```
COMPILE PROG2 COBOL LIB IN 0 4 UA "DPPE" FILE EDITFL = PATCjc
```

The program PROG2 is compiled. There are two input files. The primary symbolic file is an EDITOR file on disk, named PATCjc; it contains records to be added to, or to replace, records in the secondary file. The secondary symbolic file is on diskpack; its name and the name of its diskpack are specified on a dollar record (not shown). An output symbolic file is written to diskpack; its file and diskpack names are specified on a dollar record (not shown).

Label Equation Statement

In a multiprogramming environment, to uniquely identify source program files, you must equate the label name, using the *FILE* MCP control instruction. The internal file name and external file identifiers for COBOL files are the following:

Internal File name	External File ID	Function
CARD	CARD	Non-Editor primary symbolic file
EDITFL	EDITFL	Editor primary symbolic File
SOURCE	SOURCE	Secondary symbolic file
NEWSOU	NEWSOU	Output symbolic file
LINE	Codefile-name	Printed output listing.
PDATAB	P*mx*T*p*B	Compiler workfile, where *mx* is a 2-character code for the mix number and *p* is the processor number.

For example, to change the external file identifier on a new source language file, the label equation statement used is

?FILE NEWSOU = <file-id>

In the above example, file-id is a unique file identifier of six characters or fewer.

You can also direct the file to specific hardware devices by using the MCP FILE statement. For example, to direct the compiler workfile to a specific diskpack, use the following label equation

?FILE PDATAB = <diskpack-id>/PDATAB DPK

In the above example, diskpack-id is the name of the desired diskpack.

Note that the MCP option WRKP, when set, causes compiler workfiles to go to the system default diskpack.

For the complete information on the use of the WRKP option and the MCP FILE statement, refer to the *V Series Systems Operations Guide, Volume 1: Installation* and the *V Series Systems Operations Guide, Volume 2: System Commands*.

Source Program

The language acceptable as input to this compiler is based on the American National Standard, 1974 (ANSI-74), as described in this manual.

Input Files

The compiler has several possible input files. One or more of these source files is used to make a complete, updated COBOL program. All input files are expressed in the EBCDIC character set except when the MCP first translates them from ASCII or BCL.

Punched cards, disk, diskpack, or magnetic tape can be specified as source language input medium for a single input file or for a master file.

If you use two input files, the compiler merges them on a sequence number basis. The primary file is the file that initiates the compile and specifies that there is another input file (that is, it contains the dollar option MERGE). The other input file is the secondary file. If a matching sequence number is found in both the primary and secondary input file, the item in the primary file is used and replaces the item in the secondary file.

Records of input files are 80 bytes in length. Input from disk, diskpack, or magnetic tape can be of any blocking factor between 1 and 9 inclusive.

Library Files

COBOL source text common to several programs can be stored on disk or diskpack as source library files. One or more library files can be included in a program by use of the COPY statement. A library file can be created by any program capable of creating a file with a record length of 80 bytes, with any blocking factor between 1 and 9 inclusive. A library file is included in the logical program to be compiled, but is not included in the new source language file. A library file cannot be label equated.

A library file can contain more than one text. These portions of the library are delimited and specified by compiler control (dollar) options. Refer to "COPY Statement" in Section 6 for further information.

Compiler Intrinsics

The following are the compiler intrinsics (the *xxx* element in these intrinsic names represents the release level of the intrinsic):

- COIxxx

  Routines to handle exponentiation, blocking/deblocking, run-time errors, linage functions, task attribute functions, SORT/MERGE collating sequence, and variable length disk files.

- INDxxx

  Index sequential file-handling routines.

- RELxxx

  Relative file-handling routines.

- BSRxxx

  Routines common to both indexed and relative file handling.

- MANxxx

  Routines for handling Level 2 IPC calls.

- ISHxxx

  Routines for handling SHARED index sequential files.

- DMPRDx

  Routines for DMSII production DBP.

- SDI720
  SDM720

  Routines for using Screen Design Facility (SDF).

Dollar Options

Variations are available during the compilation process. They are activated or deactivated by dollar options. These options have a dollar sign character ($) in column 7 and can be interspersed at any point in or immediately before the primary and/or secondary input file or files. One or more dollar options can be specified following the $.

If there are no control records, the defaults are LIST, MAIN, and FEDLEVEL = 6, and a card deck is the expected input.

The dollar options SET and RESET allow the user to turn on or off, respectively, all the following options on that line without affecting any other options not specified. When SET or RESET is not the first $ option listed on the $ record, the options listed are turned on and all others turned off.

Some dollar options are considered Boolean options. A Boolean option can be set on, reset off, or set to the results of a Boolean expression. SET allows the definition of user-defined options. A user-defined option can be any name that is already an option. A user-defined option is Boolean and its default value is false.

A stack is associated with each Boolean option. Whenever a Boolean option or user-defined option is set, reset, or set to an expression value, the previous value of the Boolean option is pushed onto a stack, and the current value is set according to the Boolean expression value. A maximum of 12 user-defined options are allowed.

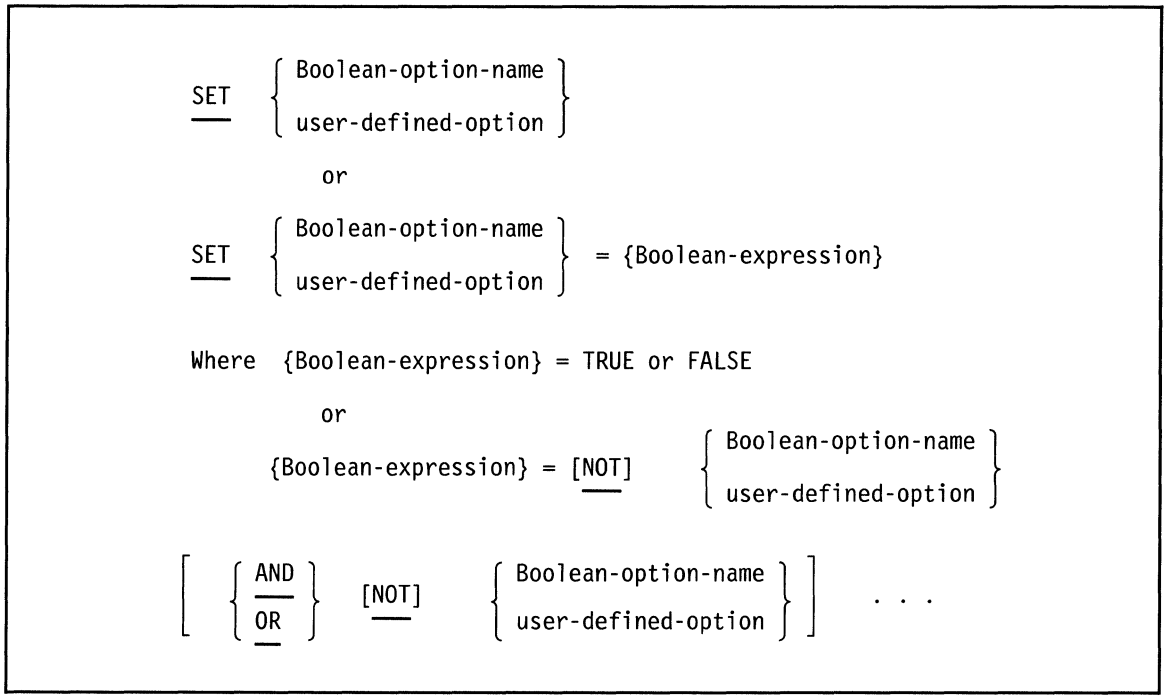
The option action indications are

$SET sets option ON.

$RESET sets option OFF.

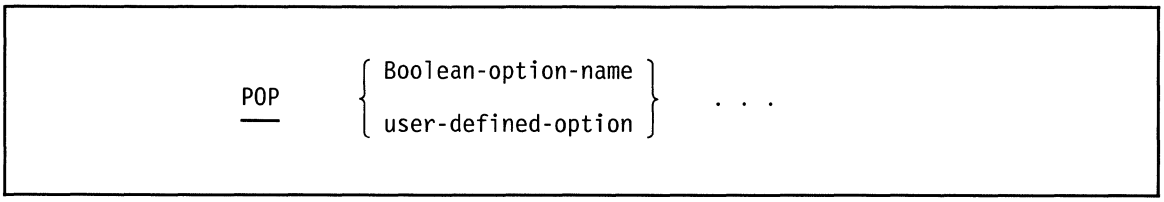
$POP sets option to a previously stacked value.

The following is the SET syntax:


SET { Boolean-option-name | user-defined-option }
or
SET { Boolean-option-name | user-defined-option } = {Boolean-expression}
Where {Boolean-expression} = TRUE or FALSE
or
{Boolean-expression} = [NOT] { Boolean-option-name | user-defined-option }
[{ AND | OR } [NOT] { Boolean-option-name | user-defined-option }] . . .

User-defined options must be declared before they can be referenced. The $ POP option discards the current setting of each option in a list of Boolean options and restores the previous value.

The following is the POP syntax:



In the following example, code appears in region-1, but not in region-2.

```
$ SET GETCODE = TRUE
   .
   .
$ SET CODE = GETCODE
   .
   .
   . (region-1)       CODE will appear
   .
$ POP CODE
   .
   .
   . (region-2)       no CODE
   .
   .
```

The available dollar options are as follows:

ANSIFILE

This option, when set, allows a variable length disk file.

The ANSIFILE option functions only when a RECORD CONTAINS clause in the FD paragraph of a disk or diskpack file is not present. In other words, this option does nothing when a RECORD CONTAINS clause is present for a disk or diskpack file or when the file resides on tape. The following applies for a disk or diskpack file.

- If the RECORD CONTAINS clause is present, it governs the file record treatment in all cases, ignoring the status of ANSIFILE. The following conditions apply:
  - If the RECORD CONTAINS clause specifies a range of record lengths, the compiler adds a 4-byte length header in front of each record created.
  - If the RECORD CONTAINS clause specifies a single length for all records in the file, the compiler does not generate the length header.
  - In either case above, the compiler checks each record length against the fixed length or length range declared in the RECORD CONTAINS clause and gives syntax error messages if they do not agree.
- If the RECORD CONTAINS clause is not present, the file record handling is determined by the presence of the ANSIFILE option. The following conditions apply:
  - If the ANSIFILE option is set and the file has variable length records declared, a 4-byte length header is added in front of each record created. Note that if ANSIFILE is set but the file contains a single length declared, no header is generated.
  - If the ANSIFILE option is absent or reset, no header is generated, regardless of whether the file has fixed or variable length records declared. In this case, the record length given in the FIB and used in all reads and writes is the length of the longest record for the file.

CHECK

Sequence errors are flagged with a warning message.

CLEAR

All options that do not have associated values are reset, except MERGE and ICM. NEW is not reset if any records have been written to the new output symbolic file.

CODE

This option lists emitted code.

CORRECTOK

Trivial errors are corrected and treated as warnings. These errors would be treated as syntax errors without this option.

CORRECTSUPR

This option performs the corrections stated under CORRECTOK and suppresses the correction warning messages.

DELETE

This option can appear only in the primary source file. It is ignored unless $MERGE is set. Images from the secondary source file are discarded (including other dollar options) until this option is reset. Items discarded are not listed in the compile listing unless the LISTDELETED option is set. Items discarded are not carried forward to the output symbolic file if $NEW is set.

DMSICM ["file-id"] [device]

This option provides a DMS ICM output file called file-id. The only device allowed is disk. If file-id is not specified, the default name provided is xxxPR1, where the xxx element is the first three characters of the database name.

Using this option, the COBOL compiler generates an ICM that can later be bound into the production DBP. This ICM is identical to the ICM generated by the use of the $ICM option except the following:

- All DMSII constructs have NTRs to the DBP.
- All STOP RUNs have NTRs to the DBP so that the database can be closed in an orderly fashion.

Note: *The only DBP opened in the DMSICM is the one to which the ICM is bound. The HIGH MEMORY clause cannot be used if DMSICM is set.*

DOUBLE

This option double spaces the listing of source images processed and error messages that were required.

ERRORLIMIT = nnn

This option terminates compilation when the number of errors specified by nnn is reached. The nnn element must be a 3-digit integer. The default value is 100.

FEDLEVEL [=] n

This option indicates the level of syntax that is to be applied to the incoming source; levels are defined by the Federal Information Processing Standards (FIPS) Publication 21-1, dated December 1, 1975. The n element must be a 1-digit integer from 1 through 6, with a default value of 6. Table 9-4 shows the meaning of each value.

Table 9-4. Meaning of Federal Level Values

Value	Meaning
1	Low level
2	Low intermediate
3	High intermediate
4	Full Federal standard
5	Full Unisys standard
6	V Series Systems extensions to Unisys standard

Specification of FEDLEVEL means that any syntax used in the source program that does not conform to the syntax allowed by the level n specified (that is, any syntax valid only at a higher level) is to be flagged with a warning message.

FEDLEVEL is intended for use when programs are written for the U.S. Federal Government. It can be used by any installation wishing to ensure that programs use only those constructs defined within some level of the COBOL ANSI-74 standard.

If FEDLEVEL is not specified in any dollar record, $ FEDLEVEL = 6 is assumed.

Table 9-5 defines these Federal levels in terms of modules of the American National Standard COBOL, ANSI X3.23-1974. The numbers in the table refer to the level within the ANSI module, and a dash in the table denotes that the corresponding module is omitted.

Table 9–5 . Federal Levels

	Federal Level			
ANSI Module	**1**	**2**	**3**	**4**
Nucleus	1	1	2	2
Table Handling	1	1	2	2
Sequential I/O	1	1	2	2
Relative I/O	-	1	2	2
Indexed I/O	-	-	-	2
Sort-Merge	-	-	1	2
Report Writer	-	-	-	-
Segmentation	-	1	1	2
Library	-	1	1	2
Debug	-	1	2	2
Interprogram Communication	-	1	2	2
Communication	-	-	2	2

ICM

The following is the ICM syntax:

```
ICM    [ "multi-file-id/file-id" ]
       [ "file-id"               ]    [ device ]
```

This option is reset by default. When set, it creates an independently compiled module (ICM) as the result of an error-free compilation. The ICM can then be bound into other programs. Multi-file-id and file-id must be alphanumeric literals, each no more than six characters long.

File-id specifies the name of the generated ICM file. If the file identifier is not present, ICM is assumed. The device can be DISK or PACK. If no device is specified, DISK is assumed.

If a compilation is intended only to produce an ICM, the dollar option MAIN must be reset.

ICMCODE

This option is reset by default. When set, it allows code for all called ICMs to be printed at the end of the listing. ICMCODE is meaningless for programs not using the CALL ... syntax or for programs with MAIN reset.

ICMFAMILY "multi-file-id"

This option when used with the CALL or CALL MODULE statement specifies a default diskpack name for ICMs within the COBOL program. Multi-file-id must be an alphanumeric literal no more than six characters long.

ICMFILE "file-id"

This option specifies a default ICM file name for the CALL MODULE statement. The option allows users to easily modify ICM names in COBOL programs.

LEV2IPC

This Boolean option provides Level 2 interprogram communication capabilities when binding at execution time is required. The default value is FALSE. When set, this option provides control transfer to programs with program names unknown at compile time. It also allows the use of the CANCEL statement.

Level 1 IPC offers greater performance than Level 2, so the Level 2 option (that is, specifying the name of the called program in a variable) should be used only when necessary.

The following are DMS considerations:

- In Level 1 IPC, the calling and the called programs must not both attempt to open the database at the same time. Doing so is equivalent to one program attempting to open the database while the same program already has it open, since Level 1 IPC creates a single program.
- In Level 2 IPC, the calling and called programs can both have the database opened simultaneously, because two separate programs are present in Level 2 IPC.

  In Level 2 IPC, if the calling program is discontinued (DS'ed), any called programs that are still present are discontinued (DS'ed).

  In Level 2 IPC, while the calling program is waiting for a called program to perform some function, the following is the status of the calling program in the mix:

  WAITING PORT READ

  If, after a called program has completed its function, the calling program is not in the active mix (for example, the calling program has been rolled out to disk), the called program waits until the calling program is again present.

A called program can be called by more than one calling program. In that case, one copy of the called program exists in memory for each calling program.

The STOQ module of the MCP is required for Level 2 IPC because that module controls port file I/O activity.

The number of parameters and the total length of the parameters must match between the calling and the called program. If they do not match, the following occurs:

- In Level 1 IPC, run time errors occur when the COBOL compiler is binding the programs together. The ODT displays the following messages:
  - For a length mismatch

    TOTAL LNGTH OF PARAMETERS DO NOT MATCH
  - For a number mismatch

    NUMB OF PARAMETERS DO NOT MATCH
- In Level 2 IPC, if the parameters do not match, the results are unpredictable.

LIBR

Dollar options in library files are processed.

LIBRFAMILY "multi-file-id"

When used with the COPY statement, this option specifies a default diskpack name for all library files within the COBOL program. Multi-file-id must be an alphanumeric literal no more than six characters long.

This option is invoked if the DISK clause is not used and a diskpack family is not explicitly specified.

LIST

This option produces a single-spaced listing of source images processed. This option is assumed unless it is reset.

LISTDELETED

This option causes deleted or voided images to be listed in the output.

LISTOMITTED

This option causes omitted images to be listed in output.

LISTP

This option lists patch images only (not main source file) when LIST is reset.

LIST$

This option lists all dollar options in output.

MAIN

When set, this option indicates to the compiler that this program is the main program and that a code file is to be produced as the result of a successful compilation. When this option is reset, only an ICM results from a successful compilation.

MAP

This option causes the compile listing to include the beginning addresses and sizes of all data and code blocks at the end of the compile listing. In addition, digit offsets of subordinate data items and Procedure Division instruction addresses are listed along the right-hand side.

Note: *If the compile statement specifies SYNTAX only, then Procedure Division addresses, data, and code block addresses are not available.*

Also note that any unreferenced data blocks are not bound into the code file and, therefore, do not appear as part of the MAP information at the end of the compile listing.

However, unreferenced data blocks are marked "NOT REFERENCED" in the DATA DIVISION of the program.

Refer to Example 6 in Appendix E for an example of MAP output.

MCS

The following is the MCS syntax:

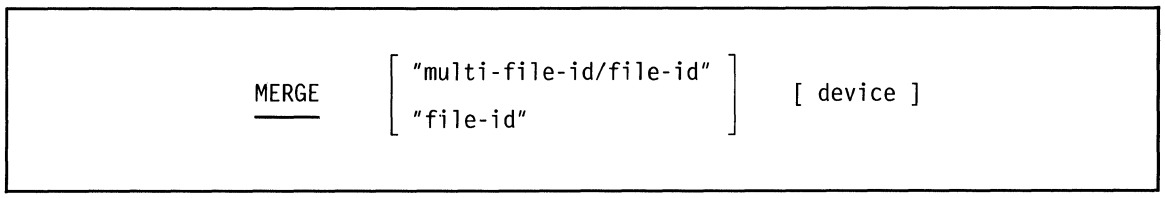
```
MCS [ "MCS-type" , [ "STOQ-input-name", "STOQ-output-name" ] ]
```

This option is used with the Screen Design Facility to specify the desired MCS (message control system) for form invocation. MCS-type can be GEMCOS (default), MSNDL, SWITCH, or CANDE. Programs running under the SWITCH

(CP3680) or VCS environments can declare MCS = MSNDL. The defaults for STOQ-input-name and STOQ-output-name are " " and "STQOUT" for GEMCOS, " " and "MCSTOQ" for MSNDL environment, and "NDSTOQ" and "SWITCH" (only valid options) for SWITCH. STOQ names will be validated for the first 6 characters.

MERGE

The following is the MERGE syntax:

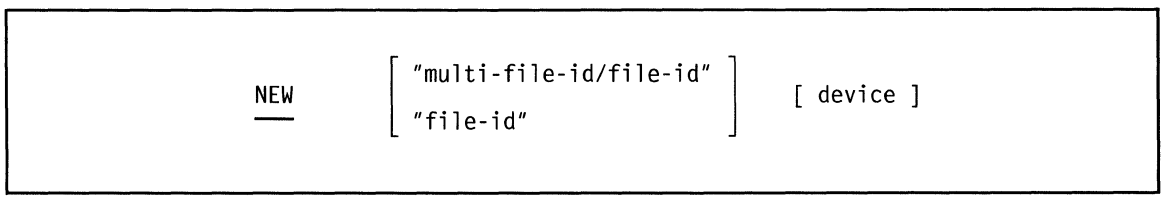


This option allows compilation of a program using two symbolic input files. The inputs are merged on the basis of sequence numbers, one input being the primary input file that contains this option. The multi-file-id and file-id must be alphanumeric literals, each no more than six characters long. File-id specifies the file name of the secondary input file. SOURCE is the assumed file name if none is specified; however, this file-id can be entirely overridden by label equation. Device is the hardware upon which the second input file resides. Valid devices are disk, tape, and diskpack with disk as the default.

This option cannot be reset after it has been set, so it remains in effect for the entire program.

NEW

The following is the NEW syntax:



This option creates a new symbolic output file. Multi-file-id and file-id must be alphanumeric literals, each no more than six characters long. File-id specifies the file name of this new output file; however, this file-id can be entirely overridden by label equation. Device is the hardware upon which the new file is to reside. Valid devices are disk, tape, and diskpack with disk as the default.

If you set the NEW option, it cannot be reset. It remains in effect for the entire program.

When the option LIST$ is set, dollar cards in the original source are written to the symbolic output file.

NEWID "literal"

This option replaces columns 73 through 80 of the output with the specified literal. The literal must be alphanumeric and no longer than 8 characters.

NOBOUNDS

This option inhibits the generation of object program code for bounds checking of subscripts and indices. This option cannot be selectively set or reset within the same program.

NORETURN

This option is used with the DMSII DELETE statement to inhibit the data transfer of the deleted record to the program work area.

OMIT

This option can appear in both primary and secondary source input files. Images from any source are ignored (but passed to a new source) until this option is reset. Items omitted are not included in the output listing unless the LISTOMITTED option is set. Any dollar options encountered while OMIT is set are processed normally.

OPTIMIZE

This option allows optimal code for arithmetic verbs to be generated without the ON SIZE ERROR phrase. When the OPTIMIZE option is set, the final resultant value is not moved to the resultant-identifier or identifiers if you do not specify the SIZE ERROR phrase and a size error condition occurs. The default value is FALSE.

PAGE

This option causes an immediate page break on the output listing.

PAGESIZE = nn

This option specifies that the maximum number of lines per page of the output listing is to be equal to nn. The nn element must be a 2-digit integer. The default value is 56.

POP

This option sets the options specified on a compiler control (dollar) record to a previously stacked value.

SAVECP

This DMS Boolean option indicates that the current path to disk is to be saved when a NOT FOUND condition occurs during a DMS operation. The default value is false.

SAVERCD

This option causes the record number of the PERFORM statement to be included as one of the stack parameters. This option is useful for run-time debugging. It is reset by default.

SEQ [nnnnnn [+nnnnnn]]

This option causes source images to be resequenced, starting with the sequence number nnnnnn and incrementing by +nnnnnn. The default start (or base) is 10, and the increment defaults to +10.

SEQCHECK

This option is the same as the CHECK option.

SEQUENCE

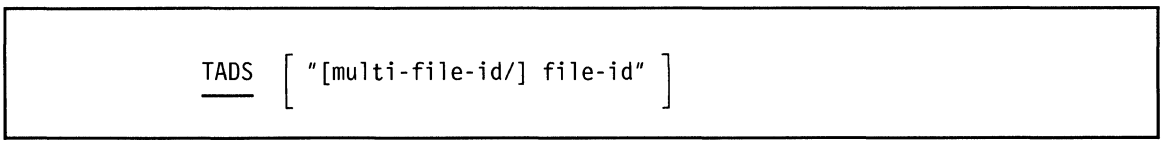
This option is the same as the SEQ option.

SUMMARY

This option causes a listing of the compiler summary when the option LIST is reset. This summary is included in the LIST option.

TADS

The following is the TADS syntax:



This option generates a program dictionary file that the IWTADS (Intelligent Workstation Testing and Debugging System) utility accesses during a COBOL debugging session. By default, this option is reset. The multi-file-id and file-id must be alphanumeric literals that do not exceed six characters in length. The multi-file-id specifies a diskpack name. The file-id specifies the name of the program dictionary file TADS generates. If you do not specify the multi-file-id, the default value is disk. If you do not specify the file-id, the default value is TADSFL.

For more information on IWE II, refer to the *V Series Intelligent Workstation Editor II (IWE II) Installation and Operations Guide.*

TITLE "literal"

This option includes the associated alphanumeric literal in compiler output headings. The default title is the program-id. Note that this option causes the compiler to store a title to be used on all subsequent pages of the listing. Therefore, it does not take effect until the next page of the listing, unless it appears as the very first source statement.

VOID nnnnnn

This option discards images from the secondary source file until the compiler encounters a sequence number larger than nnnnnn in the secondary source file. Items discarded are not included in the output symbolic file if you set $NEW. The items discarded are not included in the output listing unless you set the LISTDELETED option. This option can appear only in the primary source file. It is ignored if $MERGE is not set.

WARNFATAL

This option treats warnings as syntax errors.

WARNSUPR

This option suppresses the listing of warning messages.

XREF

This option provides a cross-reference listing of the program.

Program Output

Use the dollar option NEW to create an updated master symbolic output file, acceptable as input to the COBOL compiler. This file has a record length of 80 bytes and is blocked 9. Refer to the "NEW" dollar option in this section for information on the name and output media for these files.

Object Code

The object code can be generated to either disk or diskpack. No code is generated if syntax errors are present.

Use the STACK SIZE clause in the OBJECT-COMPUTER paragraph to control the object program stack size.

Output Listing

The source of the program input is designated on the right-hand side of the compiler output listing. EDITOR, LIBRARY, CARD, and DOLLAR records are indicated on the listings as such. In addition, digit offsets of subordinate data items and Procedure Division instruction addresses are listed along the right-hand side.

All error and warning messages are specific enough to determine the cause of the error. They are printed following the symbolic record to which they apply.

A number precedes all error messages. This number represents the count of error messages for the symbolic record. That is, the first error in a line is number 1, the second error for that line is number 2, and so forth. The number points to the position in the symbolic record at which the compiler detected an error.

Run-Time Errors

Run-time error routines supplied by the compiler, such as routines to handle out-of-range subscripts, can produce program dumps. When such a run-time error occurs, the compiler supplied routines within the program display the following message on the ODT:

ERROR AT RECORD <record-number>: <run-time error message>

The routines within the program also produce a program memory dump and abnormally terminate the program. The following is a list of the run-time messages that can be generated:

SUBSCRIPT OUT OF RANGE
MODIFYING ATTRIBUTE OF OPEN FILE
FILE MUST BE OPENED IO FOR REWRITE
FILE CANNOT BE OPEN
RECURSION NOT ALLOWED
NUMBER OF PARAMETERS DO NOT MATCH
TOTAL PARAMETERS LENGTH MISMATCH
PROGRAM MUST BE INVOKED BY OTHERS
INVALID PROGRAM IN CALL STATEMENT
PROGRAM CANNOT BE CALLED: PLEASE DS CALLER
INVALID RECORD LENGTH

If an error condition branch other than the above occurs, either of the following messages is displayed:

UNDOCUMENTED ERROR 999, or
UNDOCUMENTED ERROR 000

Time-of-Compile

A DC console message returns the time of day, in addition to the date, at which a COBOL program was compiled.

Sizes and Limits

COBOL compiler limits are shown in Table 9–6.

Table 9–6. COBOL Compiler Limits

Program Element	Maximum Allowed
Data-names + Procedure Division labels	99,999
Operands in a statement	50
Paragraph-names in a section	9,999
Section-names + FDs + 01 levels + 77 level	7,500
File named in SELECT statements	Port files 999 Non port files 999 (but only 99 can be open at once)
Source file	COBOL 90K lines COBOL (,"L") 200K lines

The COBOL compiler size as released is sufficient for all compilations. Compile speed on large programs is improved if additional main memory is given. A maximum of 500,000 bytes of main memory can be used.

An object program cannot occupy more than 1,000,000 digits (500,000 bytes) of main memory, excluding areas declared HIGH MEMORY. You can use segmentation and overlay to compress a larger program to fit within this limit. You can also realize a reduction in main memory requirements for smaller programs by use of segmentation.

COBOL compiler work files require 500,000 digits (250,000 bytes) of disk or systems memory for a 1000 line program. The requirement increases in increments of 500,000 digits per 1000 lines. COBOL (,"L") doubles the size of this work file.

Section 10
COBOL Binding

Program binding is the process of joining together parts of separately compiled programs to create a single executable program. Creating a program through binding offers several advantages over writing a program in a single piece:

- Small modules can be intellectually more manageable than large programs.
- Procedures in modules are granted direct access only to files and data items declared within the modules; this feature creates a degree of protection within the final program.
- The work of coding a large program can be subdivided among several people.
- A set of precompiled, pretested modules designed to accomplish certain functions can be established at an installation. When one of those functions is required in a program, the appropriate module can be bound into the program.
- A change to a module does not force recompilation of an entire program. Only the affected module and the main program part need be recompiled; the other precompiled modules will be bound into the final program automatically. Doing so can reduce compilation time.

The following considerations are involved in creating modular programs through binding:

- Program design must include the definition of the functions to be performed by the different modules and of the data areas to be shared by those modules. Two modules can access the same 01-level data area only if that data area has the same name in both modules and the COMMON clause has been specified in both definitions.

  COMMON data areas are the only means of passing information and parameters between modules.
- A program module is coded in much the same way as a normal COBOL program. That is, it consists of Identification, Environment, Data, and Procedure Divisions. All data items referred to in a module's Procedure Division must be defined in that module's Data Division.
- The entry points to a program module are its sections. That is, when a module is called, the calling program can specify the name of a section, and execution then begins at the first executable statement in that section; or, if a specific section is not named, execution then begins at the first executable statement in the first nondeclarative section of the called program module.

- Statements within a module can access all items declared within that program module, including procedures in other sections of that program module. The module can call other modules.
- Control is returned from a called module to its caller when an EXIT MODULE statement is executed.
- One module must be the "main" module of the final program. The *main module* is the module containing the program entry point. That is, the first executable statement in the main module is the first executable statement of the program.
- The dollar option MAIN, when set, designates that the module being compiled is the main module of a program. This setting has two effects:
  - An executable code file is created, if the compilation of the module is successful (no syntax errors are encountered). Any modules called by CALL MODULE statements are located and bound into the program. If a required ICM is not present, the compiler is suspended until the file is made available.
  - The first executable statement of the bound program is the first executable statement in the main module.
- The dollar option ICM, when set, indicates that an ICM file is to be created if the compilation is successful and assigns a name to the ICM file.

The following items should be recognized:

- The dollar option MAIN is always set by default. Therefore users writing single-module programs need not take any special actions or change any existing procedures when compiling programs. Unless MAIN is explicitly reset, a successful compile operation produces an executable code file.
- Data items to be shared among modules must be declared as common and must have the same 01-level name in each module in which they appear. The names and definitions of subordinate data items within an 01-level item are not checked; the programmer must ensure that data descriptions are correct. This instance would be most appropriate for using a COPY statement in each program to invoke the same library file containing the common data items.
- Files and buffers also can be declared as common. If the file's record area is declared as common, the record area can be accessed in more than one module.
- The same file in two different modules results in two separate files and their associated FIB, label, and buffer areas being present in the final bound program.
- If a change is made to a module, you must recompile that module to create a new ICM file, and then to compile the main module to create the new executable program.

- If the COBOL program $options are to create both an ICM and an executable code file, the program name in the compile statement must not be the same as the ICM name.

ICM files created by the COBOL compiler are not compatible with those created by any other compiler.

Examples 4 and 5 in Appendix E illustrate how COBOL binding is accomplished. The program in example 4 creates an ICM file that is then called by the program in example 5.

Appendix A
Reserved Words

The following is a list of reserved words. It includes all reserved words from the complete American National Standard (ANSI-74), and additions used with Unisys extensions. Unisys does not use all of these words at the present time, but they are all in the reserved word list for the V Series COBOL ANSI-74 compiler. The use of any of these words as a user-defined word causes an error.

A

ACCEPT
ACCESS
ACTUAL
ADD
ADVANCING
AFTER
ALL
ALPHABETIC
ALSO
ALTER
ALTERNATE
AND
ARE
AREA
AREAS
ASCENDING
ASCII
ASSIGN
AT
ATTRIBUTE
AUDIT
AUTHOR
AVAILABLE

B

BACKUP
BEFORE
BEGIN-TRANSACTION
BEGINNING
BLANK
BLOCK
BOTTOM
BY
B-2900
B-3900
B-4900

C

CALL
CANCEL
CARDS
CASSETTE
CD
CF
CH
CHANGE
CHANNEL
CHARACTER
CHARACTERS
CLOCK-UNITS
CLOSE
COBOL
CODE
CODE-SET
COLLATING
COLUMN
COLUMNS
COMMA
COMMON
COMMUNICATION
COMP
COMPUTATIONAL
COMPUTE
CONFIGURATION
CONTAINS
CONTROL
CONTROLS
COPY
CORR
CORRESPONDING
COUNT
CRCR-INPUT
CRCR-OUTPUT
CREATE
CRUNCH
CURRENCY
CURRENT

D

DATA
DATA-BASE
DATE
DATE-COMPILED
DATE-WRITTEN
DAY
DB
DE
DEADLOCK
DEBUG-CONTENTS
DEBUG-ITEM
DEBUG-LINE
DEBUG-NAME
DEBUG-SUB-1
DEBUG-SUB-2
DEBUG-SUB-3
DEBUGGING
DECIMAL-POINT
DECLARATIVES
DEFAULT
DELETE
DELIMITED

DELIMITER
DEPENDING
DESCENDING
DESTINATION
DETAIL
DISABLE
DISK
DISKPACK
DISMISS
DISPLAY
DIVIDE
DIVISION
DMERROR
DMSTATUS
DMSTRUCTURE
DOWN
DUMP
DUPLICATES
DYNAMIC

E

EGI
ELSE
EMI
ENABLE
END
END-OF-PAGE
END-TRANSACTION
ENDING
ENVIRONMENT
EOP
EQUAL
ERROR
ESI
EVERY
EXCEPTION
EXIT
EXTEND

F

FALSE
FD
FIELD
FILE
FILE-CONTROL
FILLER
FINAL
FIND
FIRST
FOOTING
FOR
FORM
FORMS
FREE
FROM

G

GCR
GENERATE
GIVING
GO
GREATER
GROUP

H

HEADING
HIGH
HIGH-VALUE
HIGH-VALUES

I

I-O
I-O-CONTROL
IDENTIFICATION
IF
IN
INDEX
INDEXED
INDICATE
INITIAL
INITIATE
INPUT
INPUT-OUTPUT
INQUIRY
INSERT
INSPECT
INSTALLATION
INTERROGATE
INTO
INVALID
INVOKE
IS

J

JUST
JUSTIFIED

K

KEY

L

LABEL
LAST
LEADING
LEFT
LENGTH
LESS
LIMIT
LIMITS
LINAGE
LINAGE-COUNTER
LINE
LINE-COUNTER
LINES
LINKAGE
LOCK
LOW-VALUE
LOW-VALUES

M

MEMORY
MERGE
MESSAGE
MODE
MODIFY
MODULES
MOVE
MULTIPLE
MULTIPLY

N

NATIVE
NEGATIVE
NEXT
NO
NO-AUDIT
NONE
NOT
NULL
NUMBER
NUMERIC

O

OBJECT-COMPUTER
OCCURS
ODT
OF
OFF
OFFER

OMITTED
ON
OPEN
OPTIONAL
OR
ORGANIZATION
OUTPUT
OVERFLOW

P

PAGE
PAGE-COUNTER
PAPERTAPE
PERFORM
PF
PH
PHASE-ENCODED
PIC
PICTURE
PLUS
POINTER
PORT
POSITION
POSITIVE
PRINTER
PRINTING
PRIOR
PROCEDURE
PROCEDURES
PROCEED
PROGRAM
PROGRAM-ID
PUNCH
PURGE

Q

QUEUE
QUOTE
QUOTES

R

RANDOM
RD
READ
READER
RECEIVE
RECORD
RECORDS
RECREATE
REDEFINES
REEL
REFERENCES
RELATIVE
RELEASE
REMAINDER
REMOTE
REMOVAL
REMOVE
RENAMES
REPLACING
REPORT
REPORTING
REPORTS
RERUN
RESERVE
RESET
RETURN
REVERSED
REWIND
REWRITE
RF
RH
RIGHT
ROUNDED
RUN

S

SAME
SAVE
SD
SEARCH
SECTION
SECURITY
SEEK
SEGMENT
SEGMENT-LIMIT
SELECT
SEND
SENTENCE
SEPARATE
SEQUENCE
SEQUENTIAL
SET
SIGN
SINGLE
SIZE
SORT
SORT-MERGE
SOURCE
SOURCE-COMPUTER
SPACE
SPACES
SPECIAL-NAMES
STACK
STANDARD
STANDARD-1
START
STATUS
STOP
STOQ-INPUT
STOQ-OUTPUT
STORE
STRING
SUB-QUEUE-1
SUB-QUEUE-2
SUB-QUEUE-3
SUBTRACT
SUM
SUPPRESS
SW1
SW2
SW3
SW4
SW5
SW6
SW7
SW8
SYMBOLIC
SYNC
SYNCHRONIZED
SYSTEM
SYSTEMERROR

T

TABLE
TAG-KEY
TAG-SEARCH
TALLYING
TAPE
TAPES
TERMINAL
TERMINATE
TEXT
THAN
THEN
THROUGH
THRU
TIME
TIMER
TIMES
TO
TODAYS-DATE

TODAYS-NAME
TOP
TRACE-OFF
TRACE-ON
TRAILING
TRANSACTION
TRANSCEIVE
TRUE
TYPE

U

UNIT
UNSTRING
UNTIL
UP
UPDATE
UPON
USAGE
USE
USING

V

VALUE
VALUES
VARYING
VIA
V-300
V-400
V-500

W

WAIT
WHEN
WHERE
WITH
WORDS
WORKING-STORAGE
WRITE

Z

ZERO
ZEROES
ZEROS
ZIP

Special Characters

+
−
*
/
**
>
<
=

Appendix B
Warning and Error Messages

COBOL recognizes two classes of errors, fatal and nonfatal. The compiler reports each instance of each error found, along with the source line that caused the error. The compiler does not produce code file if it detects an error. The compiler completes the syntax checking phase after it finds a nonfatal error. However, it does not complete this processing for fatal errors.

Fatal Errors

Nonfatal error messages are self-explanatory. Fatal errors given by the COBOL compiler are listed below.

Compiler Stack Overflow

Occasionally, a compile-time error occurs due to STACK overflow. This overflow occurs, for example, when IF or PERFORM statements are nested too deeply or there are too many levels in a data-description for the compiler to handle with its normal STACK mechanism. When the STACK overflows, you can increase its size using the following syntax:

```
COMPILE <program> WITH COBOL MEM+nnn

INSERT 40  6  mmmmmm
```

The *mmm* element is the additional stack needed for the compilation; *nnn* is the additional memory size for the compiler; *nnn* should be equal to *mmm*.

Compiler Address Error

Invalid instruction errors can occasionally occur while recovering from syntax errors. When such errors happen, fix all obvious errors and recompile.

NIL Pointer Dereference

This error occurs if the compiler attempts to access a data item using a pointer data type whose value is zero.

Uninitialized Pointer Dereference

This error occurs if the compiler attempts to access a data item using an uninitialized pointer data type.

Whenever there is a COMPILER ERROR or FATAL ERROR that cannot be resolved, forward supporting documentation to your Unisys representative.

Nonfatal Errors

The following is a list of warning and nonfatal error messages generated by the COBOL Compiler. The messages are categorized by COBOL division, program part, and compilation process.

Environment Division Error Messages

```
ACCESS MODE ALREADY SPECIFIED
ACCESS MODE CLAUSE IGNORED FOR PORT FILES
ACTUAL KEY ALREADY SPECIFIED
ACTUAL KEY IGNORED FOR DCP FILES
ALL OR NONE OF THESE FILES MUST BE IN ONE SAME RECORD AREA CLAUSE
ALTERNATE RECORD KEY CLAUSE IGNORED FOR PORT FILES
ASSIGNMENT IGNORED - FILE MUST BE ASSIGNED TO TAPE
BACKUP HARDWARE IGNORED - SYSTEM DEFAULT IS USED
CANNOT RESERVE LESS THAN 1 OR MORE THAN 99 AREAS
CHANNEL IGNORED - TAPE MUST BE 7 OR 9 CHANNEL
CHANNEL NUMBER MUST BE BETWEEN 1 AND 99
COLLATING SEQUENCE MUST BE AN ALPHABET-NAME
CURRENCY SIGN MUST BE A SINGLE CHARACTER
DEVICE TYPE NOT SPECIFIED
FEATURE NOT IMPLEMENTED
FILE MUST BE ASSIGNED TO DISKPACK
FILE STATUS IGNORED FOR DCP FILES
FILE-NAME CANNOT APPEAR IN MORE THAN ONE SAME AREA CLAUSE
FILE-NAME CANNOT APPEAR IN MORE THAN ONE SAME RECORD AREA CLAUSE
FILE-NAME CANNOT APPEAR IN MORE THAN ONE SAME SORT-MERG AREA CLAUSE
HIGH MEMORY CLAUSE NOT ALLOWED FOR PRODUCTION DBP
ICM FILE MUST BE A MASS STORAGE FILE
ILLEGAL DUPLICATE NAME
INDEXED I-O DOES NOT ALLOW ACTUAL KEY
INDEXED I-O DOES NOT ALLOW RELATIVE KEY
INDEXED I-O REQUIRES MASS STORAGE DEVICE
INVALID ACCESS MODE
INVALID ALPHABET SPECIFICATION
INVALID BACKUP DEVICE
INVALID CHARACTER SPECIFIED FOR CURRENCY SIGN
INVALID DEVICE NAME
INVALID ORGANIZATION
INVALID SAME TYPE
```

INVALID SORT-MERGE MEDIUM
INVALID TAPE KIND
KEY IS ONLY VALID FOR MASS STORAGE DEVICE
MEMORY SIZE CANNOT BE GREATER THAN 999,999
MERGING NOT IMPLEMENTED WITH DISK
MORE THAN 999 FILES DECLARED
MORE THAN 999 PORTS DECLARED
NAME CANNOT BE REFERENCED
NAME MUST BE FILE-NAME
NUMBER OF OVERFLOW TAPES MUST BE BETWEEN 3 AND 8 - 3 ASSUMED
NUMBER OF RECORDS MUST BE BETWEEN 1 AND 99,999
NUMBER OF TAPES MUST BE BETWEEN 3 AND 8 - 3 ASSUMED
ONLY ONE BUFFER USED FOR RELATIVE AND INDEXED I-O
ORGANIZATION ALREADY SPECIFIED
ORGANIZATION CLAUSE IGNORED FOR PORT FILES
PHASE ENCODED OR GCR IGNORED - MAY ONLY BE SPECIFIED WITH 9 CHANNEL
PREVIOUS ERROR PROBABLY CAUSED BY EARLIER ERRORS
READER OR PUNCH MUST BE 80 COLUMNS - 80 COLUMNS ASSUMED
RECORD KEY ALREADY SPECIFIED
RECORD KEY CLAUSE IGNORED FOR PORT FILES
RECORD KEY CLAUSE IS REQUIRED WITH INDEXED ORGANIZATION
RELATIVE I-O DOES NOT ALLOW ACTUAL KEY
RELATIVE I-O DOES NOT ALLOW RECORD OR ALTERNATE RECORD KEY
RELATIVE I-O REQUIRES MASS STORAGE DEVICE
RELATIVE I-O REQUIRES SEQUENTIAL ACCESS OR RELATIVE KEY
RELATIVE KEY ALREADY SPECIFIED
RELATIVE KEY CLAUSE IGNORED FOR PORT FILES
RESERVE AREAS CLAUSE IGNORED FOR DEVICE TYPE SPECIFIED
RESERVE AREAS CLAUSE IGNORED FOR PORTS
RESERVE CLAUSE ALREADY SPECIFIED
SAME SORT-MERG AREA CLAUSE MUST CONTAIN AT LEAST ONE SD FILE-NAME
SEGMENT LIMIT MUST BE BETWEEN 1 AND 49
SEQUENTIAL I-O DOES NOT ALLOW DYNAMIC ACCESS
SEQUENTIAL I-O DOES NOT ALLOW RECORD OR ALTERNATE RECORD KEY
SEQUENTIAL I-O DOES NOT ALLOW RELATIVE KEY
SEQUENTIAL I-O MUST HAVE SEQUENTIAL ACCESS OR ACTUAL KEY
SORT FILES CANNOT APPEAR IN A SAME AREA CLAUSE
SORTING WITH BOTH DISK AND TAPE NOT IMPLEMENTED
-SPO- HAS BEEN REPLACED BY -ODT-
STACK SIZE IN THOUSANDS MUST BE BETWEEN 1 AND 99, 3 ASSUMED
STATUS ALREADY SPECIFIED
STATUS IGNORED - NOT VALID EXCEPT ON SWITCHES
THE PBD OPTION OF MCP MUST BE RESET
UNUSED REPLACING PHRASE

Data Division Error Messages

01 LEVEL ITEM MUST FOLLOW LEVEL INDICATOR
77 LEVEL ITEM MUST BE IN WORKING-STORAGE SECTION
ACTUAL KEY MAY NOT BE SPECIFIED

ACTUAL KEY MUST BE AN ELEMENTARY ITEM
ACTUAL KEY MUST BE COMPUTATIONAL USAGE AND HAVE SIZE OF 4.
ACTUAL KEY MUST BE COMPUTATIONAL USAGE AND HAVE SIZE OF 8
ACTUAL KEY MUST BE SPECIFIED
ALL LABEL RECORDS IN LABEL CLAUSE MUST REDEFINE EACH OTHER
ALPHABET-NAME CANNOT REPRESENT LITERALS
ALPHABET-NAME REQUIRED
ALPHANUMERIC HEXADECIMAL LITERALS MUST HAVE EVEN NO. OF DIGITS
AREAS MUST BE A NUMBER BETWEEN 1 AND 100 - 100 ASSUMED
AREASIZE MUST BE A NUMBER BETWEEN 1 AND 99,999,999 - 1000 ASSUMED
AREASIZE SHOULD BE DIVISIBLE BY BLOCK SIZE - AREASIZE CORRECTED
BLANK WHEN ZERO ALREADY SPECIFIED FOR ITEM
BLANK WHEN ZERO ILLEGAL WITH INDEX USAGE
BLANK WHEN ZERO ITEM MUST HAVE NUMERIC PICTURE
BLANK WHEN ZERO NOT ALLOWED ON GROUP ITEM
BLOCK CONTAINS ALREADY SPECIFIED
BLOCK CONTAINS CLAUSE IGNORED
BLOCK CONTAINS CLAUSE IGNORED FOR PORTS
BLOCK SIZE CANNOT BE GREATER THAN 999,998
BLOCKSIZE MUST NOT BE GREATER THAN 40,000 DIGITS
BUFFER SIZE CANNOT BE GREATER THAN 999,999 DIGITS
CANNOT HAVE REDEFINES ON 01 LEVEL ITEM IN FILE SECTION
CANNOT REDEFINE ITEM WITH OCCURS CLAUSE
CANNOT REDEFINE ITEM WITH REDEFINES CLAUSE
CANNOT RENAME FD, SD, 01, 66, 77, OR 88 LEVEL ENTRY
CANNOT RENAME ITEM AT LEVEL 01 OR 77
CANNOT RENAME ITEM WHOSE SIZE IS VARIABLE
CANNOT RENAME ITEM WITH OR SUBORDINATE TO ITEM WITH OCCURS CLAUSE
CLAUSE NOT ALLOWED FOR MASS STORAGE FILES
CODE-SET CLAUSE ALREADY SPECIFIED
COMMON ALREADY SPECIFIED FOR ITEM
COMMON CLAUSE IN DATA-BASE SECTION IGNORED FOR DMCICM FILE
COMMON ONLY ALLOWED ON 01 ITEMS
COMPUTATIONAL ITEM REQUIRES NUMERIC PICTURE
CONDITION-NAME CANNOT FOLLOW GROUP ITEM WHICH IS NOT DISPLAY USAGE
CONDITION-NAME CANNOT FOLLOW INDEX DATA-NAME
CONDITION-NAME CANNOT FOLLOW LEVEL 66 ITEM
DATA BLOCK SIZE PADDED UP TO CONTROL BLOCK SIZE
DATA ITEM MUST BE ELEMENTARY DATA ITEM
DATA ITEM MUST BE UNSIGNED NUMERIC ITEM
DATA NAME NOT ALLOWED FOR THIS ATTRIBUTE
DATA RECORDS ALREADY SPECIFIED
DATA-NAME CANNOT BE IN THIS TABLE
DATA-NAME CANNOT BE TABLE
DATA-NAME MUST BE ALPHANUMERIC CATEGORY
DATA-NAME MUST BE INTEGER
DATA-NAME NOT UNIQUELY QUALIFIED
DATA-NAMES DO NOT DESCRIBE VALID RANGE FOR GROUP ITEM
DEPENDING CLAUSE CANNOT BE SUBORDINATE TO ITEM WITH OCCURS CLAUSE
DEPENDING NAME IN FIRST RECORD DESCRIPTION CANNOT BE FILLER
DEPENDING ON IS VALID ONLY FOR SEQUENTIAL I-O
DEPENDING RANGE MUST BE FROM A SMALLER TO A LARGER NUMBER

DEVICE MUST BE PRINTER IF LINAGE CLAUSE IS PRESENT
DISPLAY OR GROUP ITEM MUST BEGIN ON BYTE BOUNDARY
DUPLICATE CD
DUPLICATE FD
DUPLICATE OPTION IN LINAGE CLAUSE
DUPLICATE SD
DUPLICATE SIGN CLAUSE IN GROUP AND ELEMENTARY ITEMS
ELEMENTARY ITEM MUST HAVE PICTURE CLAUSE OR INDEX USAGE
EXTMODE MUST BE NONSTANDARD, BINARY, OR EBCDIC - EBCDIC ASSUMED
FAMILYNAME IGNORED - MUST BE A NON-NUMERIC LITERAL
FEATURE NOT IMPLEMENTED
FILE-NAME MUST BE A NON-NUMERIC LITERAL OR A DATA-NAME
FILESIZE ALREADY SPECIFIED
FILESIZE MUST BE BETWEEN 1 AND 99,999,999
FILE STATUS FOR COMMON FILE MUST BE IN COMMON DATA BLOCK
FILLER ADDED
FILLER NOT ALLOWED WITH THIS LEVEL NUMBER
FORMS ATTRIBUTE HAS BEEN REPLACED BY FORMID.
FORMS IGNORED - MUST BE TRUE OR FALSE
GROUP ITEM MAY NOT BE FILLER
GROUP ITEM MAY NOT BE SYNCHRONIZED
GROUP ITEM MAY NOT HAVE PICTURE
GROUP ITEM WITH INITIAL VALUE MUST BE DISPLAY
HIGH MEMORY ALLOWED ONLY IN WORKING-STORAGE SECTION
HIGH MEMORY CLAUSE NOT ALLOWED FOR DEFINES DATA ITEM
HIGH MEMORY CLAUSE NOT ALLOWED FOR PRODUCTION DBP
HIGH MEMORY NOT ALLOWED ON ISAM FILE
HIGH MEMORY ONLY ALLOWED ON 01 ITEMS
HOSTNAME MUST BE A NONNUMERIC LITERAL OR DATA NAME.
IGNORED - DATA NAME NOT ALLOWED FOR SORT-MERGE FILE
IGNORED - MASS STORAGE FILE CANNOT HAVE LABEL OMITTED
IGNORED UNLESS FILE IS ASSIGNED TO TAPE
ILLEGAL ASCENDING DESCENDING KEY
ILLEGAL DUPLICATE NAME
IN THRU CLAUSE, FIRST LITERAL MUST BE LESS THAN SECOND
INDEXED FILE UNREADABLE BY RELATIVE - UNRECOVERABLE
INITIAL VALUE ILLEGAL IN FILE SECTION
INITIAL VALUE ILLEGAL IN LINKAGE SECTION
INITIAL VALUE ILLEGAL WITH INDEX USAGE
INITIAL VALUE ILLEGAL WITH OCCURS
INITIAL VALUE ILLEGAL WITH REDEFINES
INITIAL VALUE ILLEGAL WITH VARYING SIZE
INITIAL VALUE NOT ALLOWED AT BOTH GROUP AND ELEMENTARY LEVELS
INVALID ATTRIBUTE FOR INDEXED OR RELATIVE FILES
INVALID CD RECORD SIZE
INVALID COMBINATION OF PICTURE CHARACTERS
INVALID ENTRY TYPE
INVALID FILE ATTRIBUTE - IGNORED
INVALID ITEM TYPE
INVALID LEVEL NUMBER
INVALID LEVEL NUMBER - LEVEL 01 ASSUMED
INVALID MAXSUBFILES VALUE-DEFAULT VALUE=1

INVALID PICTURE CATEGORY
INVALID PICTURE CHARACTER
INVALID PICTURE REPEAT COUNT
INVALID PICTURE SIGN
INVALID SIGN CLAUSE
INVALID SYMBOL
INVALID SYMBOLIC FILE ATTRIBUTE VALUE - IGNORED
INVALID USAGE
ISAM KEY CANNOT BE GREATER THAN 96 CHARACTERS
ITEM REDEFINED OR REDEFINING CANNOT BE VARIABLE SIZE
JUSTIFIED ALREADY SPECIFIED FOR ITEM
JUSTIFIED ILLEGAL WITH INDEX USAGE
JUSTIFIED NOT ALLOWED ON GROUP ITEM
KEY MUST BE A DATA-NAME
KEY MUST BE AN INTEGER
KEY MUST BE NUMERIC
KEYS MAY NOT START AT SAME LOCATION
LABEL SIZE CANNOT EXCEED 16856
LABEL TYPE ALREADY SPECIFIED
LEVEL 01 OR 77 MUST BE IN AREA A
LEVEL 01 REDEFINES SIZE GREATER THAN ORIGINAL AREA
LEVEL 01 REDEFINES SIZE LESS THAN ORIGINAL AREA
LEVEL 02 THRU 49 CANNOT FOLLOW LEVEL 66 OR 77 - LEVEL 01 ASSUMED
LITERAL TYPE MUST AGREE WITH PICTURE CATEGORY
LITERAL VALUE TRUNCATED - MUST FIT IN ITEM
LITERAL-1 MUST BE A POSITIVE INTEGER
LITERAL-2 MUST BE POSITIVE AND <= LITERAL-1
MAX RECORD LENGTH MUST BE EVEN
MAX RECORD SIZE EXCEEDS MAX BLOCK SIZE - LARGER USED
MAXIMUM DECLARED RECORD SIZE DOES NOT MATCH ACTUAL RECORD SIZE
MEANINGLESS UNLESS FILE IS ASSIGNED TO DISK
MEANINGLESS UNLESS FILE IS ASSIGNED TO DISK OR DISKPACK
MEANINGLESS UNLESS FILE IS ASSIGNED TO PUNCH
MIN RECORD LENGTH MUST BE EVEN
MINIMUM DECLARED RECORD SIZE DOES NOT MATCH ACTUAL RECORD SIZE
MORE THAN ONE LINAGE CLAUSE FOUND FOR THIS FILE
MULTIPLE DB INVOCATIONS NOT ALLOWED IN PRODUCTION DBP
MUST HAVE SIGN IS SEPARATE CLAUSE
MUST REDEFINE ITEM OF SAME SIZE
MUST REDEFINE ITEM WITH SAME LEVEL NUMBER
MUST REDEFINE LAST ITEM AT SAME LEVEL
MUST RENAME ENTRY IN PRECEDING LOGICAL RECORD
MYNAME MUST BE A NONNUMERIC LITERAL OR DATA NAME
NAME CANNOT BE REFERENCED
NAME MUST BE DISPLAY USAGE
NESTED USAGE CONFLICTS WITH GROUP USAGE
NO FD SPECIFIED
NO SD SPECIFIED
NO SELECT SPECIFIED
NOT ALLOWED FOR INDEXED FILES
NUMERIC INITIAL VALUE MUST ONLY CONTAIN DIGITS
NUMERIC OR EDITED PICTURE MAY NOT BE JUSTIFIED

OCCURS CLAUSE BOUND MUST BE BETWEEN 1 AND 999,999
OCCURS ITEM SIZE CANNOT BE GREATER THAN 999,999 DIGITS
OCCURS NOT ALLOWED WITH THIS LEVEL NUMBER
ONLY 8 DIGITS USED FOR RELATIVE KEY LENGTH
OPTIMALLENGTH IGNORED - MUST BE A NUMBER BETWEEN 0 AND 9,999
PARITY MUST BE ODD OR EVEN - ODD ASSUMED
PICTURE ALREADY SPECIFIED FOR ITEM
PICTURE FIELD MAY NOT SPECIFY MORE THAN 99,999 DIGITS
PICTURE ILLEGAL WITH INDEX USAGE
PICTURE MAY NOT SPECIFY MORE THAN 98 DIGIT POSITIONS
PICTURE OF ALL INSERTION CHARACTERS ILLEGAL
PICTURE STRING MAY NOT END WITH INSERTION CHARACTER I
PORT RECORD LENGTH TOO LONG
PROCESSOR IGNORED - MUST BE TRUE OR FALSE
PROGRAM SIZE CANNOT BE GREATER THAN 999,999 DIGITS
PROTECTION IGNORED - MUST BE ABNORMALSAVE OR TEMPORARY
READ ONLY FILE ATTRIBUTE - IGNORED
READCHECK IGNORED - MUST BE TRUE OR FALSE
RECORD CONTAINS ALREADY SPECIFIED
RECORD DEPENDING ON NAME CANNOT BE SIGNED
RECORD DEPENDING ON NAME MUST BE 4 CHARACTERS
RECORD DEPENDING ON NAME MUST BE DISPLAY USAGE
RECORD DEPENDING ON NAME MUST BE ELEMENTARY ITEM
RECORD DEPENDING ON NAME MUST BE NUMERIC
RECORD DEPENDING ON NAME MUST BE THE FIRST ITEM IN RECORD
RECORD KEY CANNOT BE VARYING SIZE
RECORD KEY MUST BE ALPHANUMERIC
RECORD KEY MUST BE IN A RECORD FOR THIS FILE
RECORD KEY OUT OF RANGE FOR VARIABLE LENGTH RECORD FILE
RECORD MUST BE SIZE SPECIFIED IN RECORD CONTAINS CLAUSE
RECORD SIZE CANNOT BE GREATER THAN 999,998
RECORD SIZE FOR READER-SORTER MUST BE EXACTLY 780 DIGITS
RECORD SIZE IS GREATER THAN MAXIMUM SPECIFIED - LARGER USED
RECORD SIZE IS LESS THAN MINIMUM SPECIFIED - SMALLER USED
RECORD SIZE MUST BE LESS THAN 40,000 DIGITS
RECORDS ARE DIFFERENT SIZES: THIS IS NOW A VARIABLE RECORD FILE
RECORDS PER BLOCK CANNOT BE GREATER THAN 999
RELATIVE KEY CANNOT BE IN A RECORD FOR THIS FILE
RELATIVE KEY MUST NOT BE SIGNED
RESERVE nn AREAS IGNORED FOR SHARED-INDEXED FILES
SAVEFACTOR IGNORED - MUST BE A NUMBER BETWEEN 0 AND 999
SIGN CLAUSE OF PARENT USED
SIGN CLAUSE REQUIRES 'S' IN PICTURE
SIGN CLAUSE REQUIRES DISPLAY OR COMP USAGE
SIGN POSITION ALREADY SPECIFIED FOR ITEM
STATUS VARIABLE CANNOT BE SPECIFIED IN COMMUNICATION SECTION
STATUS VARIABLE CANNOT BE SPECIFIED IN FILE SECTION
STATUS VARIABLE FOR PORT FILE NOT ALLOWED IN HIGH MEMORY
STATUS VARIABLE MUST BE 2 CHARACTERS
STATUS VARIABLE MUST BE ALPHANUMERIC
SUBSYSTEM IGNORED - MUST BE EITHER DSK1 - DSK8 OR DEFAULT
SYNCHRONIZED ALREADY SPECIFIED FOR ITEM

SYNCHRONIZED ILLEGAL WITH INDEX USAGE
TITLE MUST BE A NONNUMERIC LITERAL OR DATA NAME
UNDEFINED DATA-NAME
UNIDENTIFIED DATA RECORD NAME - IGNORED
USAGE ALREADY SPECIFIED FOR ITEM
USERDATA FILLER ADDED
VALUE CLAUSE ALREADY SPECIFIED FOR ITEM
VALUE CLAUSE NOT ALLOWED WITH HIGH MEMORY DATA ITEM
VALUE IGNORED FOR INDEXED FILE -- DEFAULT VALUE USED
VALUE OF BOOLEAN ATTRIBUTE MUST BE TRUE OR FALSE - FALSE ASSUMED
VALUE OF FILE ATTRIBUTE MUST BE A DATA-NAME
VARIABLE LENGTH RECORDS IGNORED - ONLY ALLOWED ON TAPE FILES
VARYING PART MUST BE LAST

Procedure Division Error Messages

A RANDOM READ MUST HAVE INVALID KEY OR A USE PROCEDURE
A RANDOM WRITE MUST HAVE INVALID KEY OR A USE PROCEDURE
A SEQUENTIAL READ MUST HAVE AT END OR A USE PROCEDURE
A START STATEMENT MUST HAVE INVALID KEY OR A USE PROCEDURE
ACCESS MODE IS INVALID FOR THIS OPERATION
ADVANCING CAN ONLY BE USED WITH SEQUENTIAL FILES
ADVANCING OPTIONS NOT ALLOWED FOR INDEXED AND RELATIVE FILES
ALL PROCEDURES MAY OCCUR ONLY ONCE IN A PROGRAM; PRIOR IGNORED
ALPHABET NAME HAS TO BE DECLARED
ALTER SUBJECT AND OBJECT SHOULD NOT BE THE SAME PARAGRAPH NAME
ARM NOT PERMITTED IN DMSICM
AT LEAST ONE OPERAND MUST BE A VARIABLE
ATTRIBUTE CANNOT BE CHANGED FOR ORGANIZATION SPECIFIED
BLOCKSIZE VALUE MAY NOT BE INCREASED, MUST BE LARGER THAN RCD-SIZE.
CANNOT BE A GROUP ITEM
CANNOT BE A VARIABLE LENGTH ITEM
CANNOT HAVE ADVANCING MNEMONIC NAME IF LINAGE CLAUSE IN FD
CANNOT HAVE MORE THAN 46 SORT-MERGE KEYS IN SAME STATEMENT
CANNOT HAVE OPERATIONAL SIGN WITHIN GROUP ITEM
CANNOT MIX TAPE AND MASS STORAGE FILES IN VARIABLE LENGTH SORT
CANNOT USE THE ALL CONSTRUCT HERE
CLAUSE NOT ALLOWED FOR MASS STORAGE FILES
CLOSE TYPE IS INVALID FOR INDEXED AND RELATIVE FILES
COMPOSITE SIZE OF OPERAND IS LARGER THAN MAX SIZE OF 98
CONDITION-NAME MUST HAVE A SINGLE VALUE
CONFLICTING USE PROCEDURE; IGNORED FOR EXTEND FILE
CONFLICTING USE PROCEDURE; IGNORED FOR INPUT FILE
CONFLICTING USE PROCEDURE; IGNORED FOR IO FILE
CONFLICTING USE PROCEDURE; IGNORED FOR OUTPUT FILE
CORRESPONDING IDENTIFIER MUST BE A GROUP ITEM
DATA ITEM MUST BE EVEN LENGTH
DATA-NAME OR LITERAL MUST BE UNSIGNED INTEGER
DEBUGGING ON ALL PROCEDURES SPECIFIED; PREVIOUS ONES IGNORED
DECLARED AS DATA-NAME, MUST NOT BE USED AS PARAGRAPH OR SECTION

DECLARED AS PARAGRAPH NAME, MUST NOT BE USED AS A SECTION NAME
DECLARED AS SECTION NAME, MUST NOT BE DECLARED AS PARAGRAPH NAME
DECLARED AS SECTION NAME, MUST NOT BE USED AS A PARAGRAPH NAME
DELIMITER LENGTH LONGER THAN DELIMITED DATA ITEM : TRUNCATED
DESCRIPTION MUST HAVE A KEY IS PHRASE
DESCRIPTION MUST HAVE AN INDEXED BY CLAUSE
DESCRIPTION MUST HAVE AN OCCURS CLAUSE
DUPLICATE DATA-NAME, NEEDS QUALIFICATION
DUPLICATE FORM NAME, NEEDS QUALIFICATION
DUPLICATE PARAGRAPH NAME DECLARATION WITHIN THIS SECTION
DUPLICATE PARAGRAPH NAMES IN OTHER SECTIONS, MUST BE QUALIFIED
DUPLICATE SECTION NAME DECLARATION
EDITING ILLEGAL
END-OF-PAGE CAN ONLY BE USED WITH SEQUENTIAL FILES
FEATURE MAY BE DEIMPLEMENTED IN FUTURE RELEASE
FEATURE NOT IMPLEMENTED
FEATURE ONLY IMPLEMENTED FOR ALPHANUMERIC LITERALS
FILE DOES NOT HAVE A LINAGE CLAUSE
FILE MUST BE A PORT FILE
FILE MUST BE ORGANIZED RELATIVE OR INDEXED
FORM NAME NOT FOUND UNDER SPECIFIED FILE NAME
FROM MUST BE FOLLOWED BY AN IDENTIFIER
HIGH MEMORY DATA ITEM NOT ALLOWED IN USING PHRASE
HIGH MEMORY DATA ITEM NOT ALLOWED IN SDF
ICM FILE NAME MUST BE SPECIFIED IN $ICMFILE OPTION
IDENTIFIER MUST BE AN 01 LEVEL ITEM
IDENTIFIER MUST BE ELEMENTARY NUMERIC INTEGER
ILLEGAL DATA-NAME TYPE
ILLEGAL FOR USASI LABELS, NO SERIALNO FIELD IN LABEL DEFINITION
ILLEGAL MNEMONIC ATTRIBUTE VALUE
ILLEGAL MOVE ITEM
ILLEGAL OPERAND DESCRIPTION
ILLEGAL SET ITEM
ILLEGAL STOP STATEMENT
ILLEGAL SUBSCRIPTING TYPE
ILLEGAL VARYING CLAUSE
INCOMPLETE CONDITIONAL EXPRESSION
INDEX NOT ASSOCIATED WITH TABLE
INDEXING AND SUBSCRIPTING COMBINATION NOT ALLOWED
INPUT MUST BE CONDITION-NAME
INVALID ATTRIBUTE FOR THIS DEVICE TYPE
INVALID CLOSE
INVALID DATA NAME
INVALID FILE ATTRIBUTE FOR THE WAIT VERB
INVALID FILE-CONTROL SPECIFICATIONS
INVALID FORMAT FOR PORT FILES
INVALID I/O TYPE
INVALID KEY CAN ONLY BE USED WITH RANDOM FILES
INVALID KEY OPTION IS INVALID FOR SEQUENTIAL FILES
INVALID KEY OPTION IS INVALID FOR SEQUENTIALLY ACCESSED FILES
INVALID KEY OR USE PROCEDURE MUST BE SPECIFIED FOR DYNAMIC READS
INVALID PORT FILE ATTRIBUTE

INVALID PORT FILE I/O TYPE
INVALID READER SORTER EXCEPTION NAME
INVALID SUBPORT ATTRIBUTE
INVALID SYNTAX
INVALID USE OF ALTER STATEMENT
IO CLAUSE MUST REFER TO A MASS STORAGE FILE
ISAM FILES CANNOT BE INVOKED IN PRODUCTION DBP
ITEM CANNOT BE ALPHABETIC
ITEM CANNOT BE NUMERIC
ITEM CANNOT BE SUBSCRIPTED
ITEM MUST BE INDEX NAME
ITEM MUST BE UNIQUELY DECLARED IN THE LINKAGE SECTION
ITEM NOT DECLARED
KEY CLAUSE IS NOT ALLOWED FOR SEQUENTIAL ACCESS FILES
KEY SPECIFIED IS NOT A DECLARED KEY FOR THIS FILE
LABELSIZE VALUE MAY NOT BE INCREASED
LEFT OPERAND AND RIGHT OPERAND MUST HAVE THE SAME USAGE
LEFT OPERAND MUST BE INDEXED BY THE FIRST INDEX NAME IN TABLE
LENGTH MUST BE 6 DIGITS
LEVEL 2 IPC RESET BY DOLLAR OPTION, CALL IDENT NOT ALLOWED
LEVEL 2 IPC RESET BY DOLLAR OPTION, CANCEL STMT NOT ALLOWED
LEVEL-88 CONDITION NAME NOT ALLOWED FOR DEBUGGING
LINAGE CLAUSE MUST BE IN FD FOR END-OF-PAGE OPTION
LINAGE-COUNTER NEEDS TO BE QUALIFIED
LINAGE-COUNTER QUALIFIER MUST BE A FILE NAME
LINE OR LINES IGNORED; CANNOT FOLLOW MNEMONIC NAME
LINKAGE ITEM MUST BE IN PROCEDURE DIV. USING CLAUSE
LINKAGE SECTION NOT SEEN
LITERAL MUST NOT BE ZERO
LOCK OPTION ON OPEN I-O WILL BE OVERRIDDEN TO "LOCK" ON ASR 6.7
MAXIMUM SORT/MERGE KEY SIZE IS 98 UNITS
MAY BE REFERENCED ONLY WITHIN A DEBUGGING SECTION
MEMORY SIZE TOO LARGE, IGNORED
MISSING PROCEDURE NAME
MNEMONIC NAME MUST BE ODT
MOVE ITEMS NOT COMPATIBLE
MULTI USING FILES NOT ALLOWED FOR SORT MERGE ON DISK
MUST BE A DISPLAY DATA-NAME
MUST BE A DISPLAY DATA-NAME OR ALPHANUMERIC LITERAL
MUST BE A FILE NAME
MUST BE A MASS STORAGE FILE
MUST BE A NUMERIC DATA-NAME
MUST BE A RECORD NAME
MUST BE A SORT-MERGE FILE NAME
MUST BE AN ELEMENTARY NUMERIC DATA-NAME OR LITERAL
MUST BE DATA-NAME
MUST BE DECLARED AS CD NAME
MUST BE ELEMENTARY NUMERIC DATA-NAME
MUST BE NONNUMERIC LITERAL OR ALPHANUMERIC DATA ITEM
MUST BE NONUMERIC LITERAL
MUST BE NUMERIC LITERAL
MUST BE TYPE INTEGER

MUST BE TYPE INTEGER OR A CHANNEL NUMBER
MUST BE UNSIGNED
MUST BE VARIABLE LENGTH RECORD FILE
MUST SPECIFY A SUBPORT INDEX
MUST SPECIFY ACTUAL KEY FOR SHARED FILES
NAME MAY OCCUR ONLY ONCE IN USE FOR DEBUG STATEMENT; PRIOR IGNORED
NEED AT LEAST 2 SENDING ITEMS
NEXT IS NOT VALID FOR THIS ACCCESS OR ORGANIZATION
NO ACTUAL KEY CLAUSE SPECIFIED
NO CORRESPONDING ITEMS
NO LABEL BY THIS NAME DECLARED IN SECTION USED AS QUALIFIER
NO LINAGE CLAUSE FOR ANY FILE, SO NO LINAGE-COUNTER
NO PARAGRAPH OR SECTION BY THIS NAME WAS DECLARED IN PROGRAM
NO REWIND DISMISS IGNORED: MUST REFER TO A TAPE FILE
NO REWIND IGNORED: MUST REFER TO A TAPE FILE
NO SUCH FIELD IN THIS RECORD
NONSTANDARD USAGE WILL BE DISALLOWED IN FUTURE RELEASE
NOT A VALID CONDITION-NAME
NOT A VALID OPEN MODE
NOT IMPLEMENTED FOR SORT MERGE
NUMERIC LITERAL MUST BE UNSIGNED OR BE SIGNED POSITIVE
OPEN EXTEND IS NOT VALID WITH MULTIPLE FILE
OPEN EXTEND NOT ALLOWED FOR INDEXED AND RELATIVE FILES
OPEN TYPE IS INVALID FOR INDEXED AND RELATIVE FILES
OPERAND MUST BE AN INTEGER
OPERAND MUST BE ELEMENTARY NUMERIC DATA-NAME OR NUMERIC LITERAL
OVER 8 SORT MERGE USING FILES
OVERLAPPING DATA
PARAGRAPH MUST CONSIST OF SINGLE GO TO STATEMENT
PROCEDURES WITHIN DEBUGGING SECTION MAY NOT APPEAR IN USE FOR DEBUG
PURGE IGNORED: MUST REFER TO A TAPE OR MASS STORAGE FILE
QUALIFICATION NOT COMPLETE
READ ONLY ITEM (MAY BE REFERENCED BUT NOT CHANGED)
RECEIVING FIELD LENGTH IS TOO SMALL
RECEIVING FIELD TRUNCATION
RECEIVING IDENTIFIER MUST BE AN ELEMENTARY ITEM
RECEIVING IDENTIFIER MUST BE NUMERIC OR NUMERIC EDITED
RECORD MUST BE SAME SIZE AS SORT-MERGE RECORD
RECORD SIZE MUST MATCH SIZE OF RECORD IN SORT OR MERGE FILE
RECORD-NAME AND IDENTIFIER CANNOT REFER TO SAME STORAGE AREA
RECORDS IN INPUT FILE CANNOT HAVE VARIOUS SIZES
REEL OR UNIT IGNORED: MUST REFER TO A SEQUENTIAL FILE
REEL OR UNIT IGNORED: MUST REFER TO A TAPE FILE
REFERENCED AS SECTION NAME, MUST NOT BE USED AS A PARAGRAPH NAME
REMINDER: THIS STATEMENT MUST BE ALTERED PRIOR TO ITS EXECUTION
REMOVE CRUNCH MUST REFER TO A MASS STORAGE DEVICE
REMOVE IGNORED: MUST REFER TO A MASS STORAGE FILE
REVERSED OR NO REWIND IGNORED: MUST REFER TO A SEQUENTIAL FILE
REVERSED OR NO REWIND IGNORED: MUST REFER TO A TAPE FILE
RIGHT OPERAND CANNOT BE INDEXED BY THE FIRST INDEX NAME IN TABLE
RIGHT OPERAND CANNOT BE REFERENCED IN THE KEY CLAUSE OF THE TABLE
SAVE OR CRUNCH IGNORED: MUST REFER TO A MASS STORAGE FILE

SECTION CANNOT BE TWO DIFFERENT TYPES OF IMPLICIT PROCEDURES
SEEK REQUIRES A FILE WITH AN ACTUAL KEY
SEGMENT NUMBER MUST BE UNSIGNED INTEGER LESS THAN OR EQUAL TO 999
SENDING FIELD IN COMPUTE STATEMENT MUST BE ARITHMETIC EXPRESSION
SENDING ITEM IS NOT A BOOLEAN ITEM
SET ITEMS NOT COMPATIBLE
SHARED ATTRIBUTE CANNOT BE CHANGED FOR INDEXED FILES
SHARED FILES MUST BE RANDOM ACCESS
SHARED FILES REQUIRE USE PROCEDURE OR ON OVERFLOW CLAUSE
SIZE MUST BE ONE DIGIT LONG
SIZE OF OPERAND IS LARGER THAN MAX SIZE OF 98
SIZE OF REPLACING ITEM MUST EQUAL 1 TO REPLACE 1 CHARACTER
SIZE OF REPLACING ITEM MUST EQUAL SIZE OF ITEM REPLACED
SORT KEY OUT OF RANGE: NOT WITHIN SHORTEST RECORD FOR THIS FILE
SORT-MERGE KEY NOT IN APPROPRIATE SORT-MERGE FILE
-SPO- HAS BEEN REPLACED BY -ODT-
SUBSCRIPT CANNOT BE 0
SUBSCRIPT LARGER THAN MAXIMUM IN OCCURS CLAUSE
SUBSCRIPT LONGER THAN 6 DIGITS
THE AT END PHRASE CAN ONLY BE FOR SEQUENTIAL FILES
THE FILE ACCESS MODE MUST BE RANDOM OR SEQUENTIAL
THE INVALID KEY PHRASE CAN ONLY BE FOR RANDOM FILES
THE KEY CLAUSE OR A NEXT MUST BE SPECIFIED FOR SEQUENTIAL FILES
TOO FEW SUBSCRIPTS
TOO MANY SUBSCRIPTS
TYPE CANNOT BE MNEMONIC
TYPE MUST BE MNEMONIC
TYPE MUST NUMERIC OR ALPHANUMERIC
UNRECOGNIZED SDF FILE NAME
UNRECOGNIZED SDF FORM NAME
USAGE MUST BE COMPUTATIONAL
USAGE MUST BE DISPLAY
USAGE MUST BE DISPLAY OR COMPUTATIONAL
USE PROCEDURE IGNORED; MISSING LABEL DEFINITION
USING FIELD TRUNCATED TO 100 UNITS
VALUE MUST BE TYPE ALPHANUMERIC
VALUE MUST BE TYPE BOOLEAN
VALUE MUST BE TYPE NUMERIC
VALUE OF FILESIZE MUST BE SPECIFIED

Syntax Error Messages

AREA A OF CONTINUATION LINE MUST BE BLANK
ATTRIBUTE STACK TAG MISMATCH
BOOLEAN OPTION EXPECTED
BOOLEAN OPTION STACK OVERFLOW
BOOLEAN OPTION STACK UNDERFLOW
CHECKID SET - COLUMNS 73-80 CONTAIN NON-SPACE CHARACTERS
COBOL WORD CONTAINS SPECIAL CHARACTER
DEVICE(S) ARE UNKNOWN

DIVISION HEADER IS NOT IN AREA A
DIVISION HEADER MISSING
EQUAL SEQUENCE NUMBERS ON SOURCE LINES
ERRORLIMIT EXCEEDED, COMPILE TERMINATED
EXCEEDED MAXIMUM ALLOWABLE USER DEFINED DOLLAR CARD OPTIONS
EXPECTED DOLLAR CARD OPTION
EXPECTED NUMBER FOR DOLLAR CARD OPTION
FATAL ERROR: DEVICE(S) ARE UNKNOWN
FD LEVEL INDICATOR EXPECTED
FILE-NAME, PACK-NAME MAY BE NO LONGER THAN 6 CHARACTERS
FILE-NAME, PACK-NAME MUST BE A COBOL WORD
FLUSHED TO END OF DIVISION
HEXADECIMAL LITERAL CONTAINS NON-HEXADECIMAL CHARACTER
HEXADECIMAL LITERAL MUST CONTAIN AT LEAST ONE CHARACTER
ICM OPTION REQUIRES MASS STORAGE - DISK ASSUMED
IDENTIFICATION DIVISION MISSING
IGNORED INVALID SYMBOL
ILLEGAL INVOCATION SYNTAX
IN OR OF KEYWORD EXPECTED
INSERTED MISSING <symbol>
INTEGER EXCEEDS 18 DIGITS
INVALID A FIELD CHARACTER
INVALID ATTRIBUTE NAME, REPLACED BY ACCESSMODE
INVALID ATTRIBUTE VALUE
INVALID CD SYNTAX
INVALID COBOL CHARACTER
INVALID DMSICM NAME
INVALID DMSTATUS NAME, REPLACED BY DMERROR
INVALID DUPLICATE NAME
INVALID FILE NAME ON DOLLAR CARD OPTION
INVALID INSERT VALUE FOR CARD DEVICE
INVALID INSERT VALUE FOR COMPILE TYPE
INVALID INSERT VALUE FOR NEWSOU DEVICE
INVALID INSERT VALUE FOR SOURCE DEVICE
INVALID LEVEL NUMBER
INVALID SYMBOL IN COLUMN 7
INVALID SYNTAX
INVALID SYNTAX IN ATTRIBUTE STATEMENT
INVALID SYNTAX IN DMSTATUS STATEMENT
INVALID SYNTAX ON DOLLAR CARD OPTION
INVALID SYNTAX: EXPECTED <name>
INVALID SYNTAX: EXPECTED <symbol>
INVALID TO START NEWSOURCE DOLLAR OPTION
LABEL DECLARATION NOT EXPECTED: WORD IN AREA A MOVED TO AREA B
LIBRARY TEXT NAME NOT FOUND: CHECK SPELLING
LITERAL MUST BE FOLLOWED BY A SPACE),;.
LITERAL MUST BE PRECEDED BY A SPACE OR A (
MERGE OR NEW STARTED, CANNOT BE CHANGED
MISSING 'BY' IN COPY REPLACING STATEMENT
MISSING CLOSING QUOTE
MISSING DATA-BASE SECTION
MISSING DIVISION NAME

MISSING HEXADECIMAL LITERAL CONTINUATION @
MISSING HEXADECIMAL LITERAL ENDING @
MISSING NAME
MISSING NONNUMERIC LITERAL CONTINUATION QUOTE
MISSING NONNUMERIC LITERAL ENDING QUOTE
MISSING PERIOD
MISSING PICTURE STRING
MISSING SPACE AFTER ARITHMETIC OR RELATIONAL OPERATOR
MISSING SPACE OR SEPARATOR AFTER WORD OR NUMBER
NESTED COPY STATEMENT IS INVALID
NON-NUMERIC SEQUENCE NUMBER, CHARACTERS REPLACED BY 0
NONNUMERIC LITERAL MUST CONTAIN AT LEAST ONE CHARACTER
NULL QUOTED STRING
NUMERIC ITEM MUST CONTAIN AT LEAST ONE DIGIT
NUMERIC LITERAL CONTAINS NON-NUMERIC CHARACTER
NUMERIC LITERAL WAS EXPECTED
NUMERIC TOKEN: OUT OF RANGE FOR DOLLAR CARD OPTION
OPTION IGNORED - DOLLAR CARD SPECIFIED EARLIER
OVERFLOWED SYNTAX STACK
PERIOD EXPECTED
PERIOD OR ALL EXPECTED
PREMATURE EOF
PREVIOUS SYNTAX ERROR
PROGRAM-NAME MISSING
PSEUDO-TEXT-1 MAY NOT BE EMPTY
PSEUDO_TEXT, LITERAL, IDENTIFIER, OR COBOL WORD EXPECTED
REPLACED INVALID SYMBOL BY <name>
REPLACING ITEM MAY BE ONE WORD, IDENTIFIER, OR LITERAL ONLY
REPLACING PHRASE WAS NOT REPLACED: CHECK SPELLING
RESPELLED INVALID SYMBOL BY <name>
RESUMED PARSING HERE
SDF DIRECTORY INVALID
SDF DIRECTORY NOT FOUND
SDF FIELD NOT FOUND
SDF FILE IN USE
SDF FILE INVALID
SDF FILE NAME EXPECTED
SDF FILE NOT FOUND
SDF FILE NOT PRESENT
SDF FORM NAME EXPECTED
SDF FORM NOT FOUND
SEQUENCE NUMBER OVERFLOW, SEQUENCING TERMINATED
SOURCE LINES OUT OF NUMERIC SEQUENCE
SYMBOL TOO LONG TRUNCATED
TOKEN INSERTED NOT FOUND
TOO MANY OPERANDS - REDUCE THE NUMBER
UNDECLARED DOLLAR CARD OPTION
UNIMPLEMENTED FEATURE
UNKNOWN ITEMS IN IDENTIFICATION DIVISION
UNMATCHED PARENTHESES ON DOLLAR CARD
UNRECOGNIZED DOLLAR CARD OPTION IGNORED

UNRECOGNIZED MCS TYPE
WORD MAY NOT END WITH A HYPHEN

Code Generation Phase Error Messages

ALL REGISTERS BUSY
ALTERING PARAGRAPHS IN INDEPENDENT SEGMENTS IS NOT IMPLEMENTED
CODE BLOCK MUST BE READ BEF CODE TOKENS
DATA BLOCK MUST BE READ BEF DATA TOKENS
DEBUGGED NAME NOT FOUND
DMSTATUS FUNCTIONS UNAVAILABLE WITHOUT DATA BASE
EDITED RECEIVING FIELD EXCEEDS 160 CHARACTER MAX
EDITED RECEIVING FIELD TOO LONG FOR SENDING FIELD LENGTH
EXPECTED CONDITIONAL DATA-NAME TYPE
EXPONENTIATION RESULT OVERFLOW
FATAL ERROR DISCOVERED IN BIND PHASE
FUNCTION NOT IMPLEMENTED
ICM BLOCK TABLE OVERFLOW
ICM FILE NOT AVAILABLE
ICM HEADER MUST BE READ FIRST
ICM NOT OPEN
ICM PAGE AND TABLE TYPE NOT COMPATIBLE
ICM PUT ERROR
ICM TOKEN TABLE OVERFLOW
ILLEGAL ICM TOKEN
INCORRECT ICM FORMAT
INVALID ICM PAGE NUMBER
INVALID RELATIONAL OPERATOR
INVALID SUBPORT INDEX
ISAM FILES CANNOT BE INVOKED IN PRODUCTION DBP
LINEAR SEARCH
MAXIMUM LENGTH OF EDITED MASK EXCEEDED
MAXIMUM OF 15 OPERANDS ALLOWED
MAXIMUM NUMBER OF SECONDS TO WAIT EXCEEDED
MEMORY SIZE TRUNCATION
MISSING DYADIC OPERATOR
MISSING ICM BLOCK DESCRIPTOR
MISSING ICM HEADER
MISSING ICM MODULE HEADER
MISSING PARAGRAPH NAME AS SUBJECT OF GO TO STATEMENT
NOT IMPLEMENTED ICM ERROR
OBJECT FILE NOT GENERATED
OPERAND STACK OVERFLOW
OPERAND STACK UNDERFLOW
OPERATOR STACK OVERFLOW
OPERATOR STACK UNDERFLOW
PRIMARYSTORAGE WILL BE 8 DIGITS ON FUTURE RELEASES
RECEIVING FIELD LENGTH IS TOO SMALL
RECEIVING FIELD TRUNCATION
RELATIVE INDEXING CAN BE USED ONLY WITH AN INDEX-NAME

SET SELECTION EXPRESSION LIMIT EXCEEDED
SIZE LIMIT EXCEEDED
TOO MANY OPERANDS - REDUCE THE NUMBER
UNKNOWN ICM ERROR
USE ICM PUT CODE TOKEN
USE ICM PUT DATA TOKEN
VALUE OUT OF RANGE

DMSII Error Messages

AUDIT MUST BE DDF SPECIFIED
COBOL NUMBER SIZE EXCEEDED
DATABASE NAME EXPECTED
DATABASE, DATASET OR REMAP NAME EXPECTED
DATASET FOR THIS SET NOT INVOKED
DATASET OR REMAP NAME EXPECTED
DATASET VIA EXPECTED WITH THIS SET
DATASET, REMAP OR SET NAME EXPECTED
DATASET, REMAP, SET, OR DATABASE NAME EXPECTED
DDF IS NOT A COMPATIBLE VERSION FOR COBOL - RECOMPILE DASDL
DISJOINT DATASET, REMAP OR SET NAME EXPECTED
DM IDENT EXPECTED
DM IDENT IS RESERVED WORD
DM IDENT LENGTH EXCEEDED
GLOBAL MUST BE DDF DECLARED
ILLEGAL DB INVOKE
ILLEGAL LEVEL 01 INVOKE
ILLEGAL NAMING SERIES
ILLEGAL NONE PHRASE
ILLEGAL USING PHRASE
INCOMPATIBLE KEY VALUE TYPE
INTERFACE DESCRIPTION MISMATCH
INVALID COMBINATION OF STRUCTURES
INVALID DATABASE NAME
INVALID DDF
INVALID ORGANIZATION
INVALID SYNTAX
KEY NAME EXPECTED
LEVEL 01 EXPECTED
LOGICAL DATABASE NAME EXPECTED
MAX DM FILES IN SIB EXCEEDED
NO OPERATION ACROSS DB INVOKES
NO SETS ARE ASSOCIATED WITH THE USING SET ID
NOT A DATABASE STRUCTURE
NOT A DISJOINT STRUCTURE
POSSIBLE DM KEY VALUE TRUNCATION
RELATED SET NAME EXPECTED
REMAP OR DATASET NOT IN THE INVOKED DATABASE
RESTART TYPE DS EXPECTED
SET INVOKED IS NOT IN SET LIST FOR REMAP OR DATASET

```
SET NAME EXPECTED
SET SELECTION EXPRESSION LIMIT EXCEEDED
SET TO NULL EXPECTS ALPHANUM OR NUMERIC DATA-NAME
SUBSET-NAME MUST BE A MANUAL SUBSET
THIS SET HAS NO KEYS
UNIMPLEMENTED DM CONSTRUCT
VARIABLE FORMAT ERROR
```

Federal Level Warnings

```
FEDERAL LEVEL CHECKING - CONSTRUCT IS LOW LEVEL
FEDERAL LEVEL CHECKING - CONSTRUCT IS LOW INTERMEDIATE LEVEL
FEDERAL LEVEL CHECKING - CONSTRUCT IS HIGH INTERMEDIATE LEVEL
FEDERAL LEVEL CHECKING - CONSTRUCT IS HIGH LEVEL
FEDERAL LEVEL CHECKING - CONSTRUCT IS UNISYS CORP. STANDARD
FEDERAL LEVEL CHECKING - CONSTRUCT IS MEDIUM SYSTEMS EXTENSION
FEDERAL LEVEL CHECKING - CONSTRUCT IS IMPLEMENTOR NAME
```

Compiler Error Messages

```
ARRAY SIZES NOT EQUAL
ARRAY SIZES OUT OF RANGE
ASSERTION FAILED
CASE SELECTOR OUT OF RANGE
COERCION PATH AMBIGUOUS
CONVERSION FAILED
HEAP AND STACK COLLISION
HEAP OVERFLOW
HEX STRING LENGTHS NOT EQUAL
INVALID BUFFER USED FOR BUFIO
MUST CALL PREP_USERDEF_BUFIO
MUST CALL USERDEF_BUFFER_IO
NO PROC RESULT
NON-NUMERIC STRING
PORT BUFFER SIZE NOT MOD 2
STRING INDEX RANGE
STRING LENGTH RANGE
SUBSCRIPT RANGE
TRUNCATION FAILED
VALUE OUT OF RANGE
VF: CREATE LENGTH EXCEEDS 1800 DIGITS
VF: CREATE LENGTH MUST BE MOD 4
VF: ELEMENT SIZE CANNOT EXCEED 100 DIGITS
VF: FILE ALREADY CREATED
VF: FILE NOT CREATED
VF: FILE OVERFLOW
VF: KEY SIZE CANNOT BE ZERO OR EXCEED 100 DIGITS
VF: RECORD NUMBER CANNOT BE ZERO
VF: RECORD NUMBER CANNOT EXCEED NUMBER OF RECORDS WRITTEN
VF: TOO MANY FILES
```

```
VF: TOO MANY RECORDS
VF: WORK AREA LENGTH MUST EQUAL CREATED LENGTH
```

Binding Phase Error Messages

```
A BLOCK NUMBER > 9999 FOUND IN ICM
ADDRESS ERROR
BCT OVER 300K
FIB ADDRESS GREATER THAN 400 KD
HIGH MEMORY SIZE EXCEEDS ALLOWABLE LIMIT
INCOMPATIBLE ADDRESS EXTENSION
INCOMPATIBLE INTRINSICS OR ICM MODULE <icm-name>
INCORRECT OVERLAY SPECIFICATION
INCORRECT REFERENCE TO EXTERNAL TABLE:
IN PROGRAM
INSTRUCTION TIME OUT
INVALID DEFN. OF IHEAP DATA_BLOCK - MUST BE 18 DIGITS
INVALID INSTRUCTION
INVALID KEY ON READ CODE STREAM
INVALID KEY ON WRITE CODE STREAM
INVALID TOKEN TYPE
LABEL ADDRESS EXCEEDS 999,999
MEMORY PARITY
MEMORY SIZE >= 300K DIGITS REQUIRED
```

To resolve this situation, specify *MEMORY SIZE 150000 CHARACTERS.* The compiler will then determine the proper memory size.

```
MORE THAN 999 FILES DECLARED
NAME CONFLICT OR ICM INCOMPATIBLE <icm-name>
NEED TO SPECIFY AN EXTEND IN THE BIND DECK
NIL POINTER DEREFERENCE
NO HEADER IN ICM
NON-EXTENDED BRANCH ADDRESS EXCEEDS 300KD
NON-EXTENDED DATA ADDRESS EXCEEDS 100KD
NUMBER OF USER_DEFINED SEGMENTS EXCEEDS 999
ONLY ONE COMMON DATA BLOCK HAS HIGH MEMORY SET
PROGRAM ENTRY POINT IS A DUP BLOCK
PROGRAM ENTRY POINT IS NOT A CODE BLOCK
PROGRAM MAY NEED HEAPSPACE: IS NOT DECLARED
PROGRAM SIZE >= 1000K DIGITS
REFERENCE PARAMETER LIST IS
SHARED FILE ADDRESS >= 400K DIGITS
SHARED FILE FIB ADDRESS EXCEEDS 400KD
STACK OVERFLOW
SYMBOL_FILE OVERFLOW - TOO MANY IDENTIFIERS
THE FOLLOWING BLOCKS ARE UNDEFINED <name>
TROUBLE ON CODEFILE
TWO COPIES OF DATA BLOCK WITH DIFFERENT SIZES FOUND
UNDEFINED CODE TOKEN TYPE FOUND IN ICM
UNINITIALIZED POINTER DEREFERENCE
UNRESOLVED DATA BLOCK
```

Appendix C
EBCDIC, ASCII, and BCL Reference Tables

The charts given in this appendix define the EBCDIC, ASCII, and BCL code specifications as implemented by Unisys for V Series processors.

Tables C-1 and C-2 show the USASCII X3.4–1963 and USASCII X3.4–1967 code sets, respectively. The major differences in the code sets are in columns 0 and 1 (control characters) and columns 6 and 7 (lowercase characters).

Table C-3 reflects the Extended Binary Coded Decimal Interchange Code (EBCDIC). Each box contains the graphic representation above and the punch card below.

Table C-4 reflects the BCL Language. This BCL is a 6-bit tape code.

In each of these tables, the high-order bit is either b8 for EBCDIC, b7 for USASCII, or b6 for BCL. The row number is the decimal equivalent of the binary number formed by the four lowest-order bits: b4, b3, b2, and b1. The column number is formed by the remaining bits: b8, b7, b6, and b5 for EBCDIC, b7, b6, and b5 for USASCII; b6 and b5 for BCL.

For example, the uppercase character *P* has a bit representation as follows:

- In USASCII (Tables C-1 and C-2), it is located in column 5, row 1, which is the following:

b_7	b_6	b_5	b_4	b_3	b_2	b_1
1	0	1	0	0	0	0

- In EBCDIC (Table C-3), it is located in column 13, row 2, which is the following:

b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1
1	1	0	1	0	1	1	1

- In BCL (Table C-4), it is located in column 2, row 7, which is the following:

b_6	b_5	b_4	b_3	b_2	b_1
1	0	0	1	1	1

Table C-1. USASCII X3.4–1963

Bits b_4 b_3 b_2 b_1	Row ↓ / Column →	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
		0	1	2	3	4	5	6	7
0 0 0 0	0	NULL	DC0	b	0	@	P		
0 0 0 1	1	SOM	DC1	!	1	A	Q		
0 0 1 0	2	EOA	DC2	,,	2	B	R		
0 0 1 1	3	EOM	DC3	#	3	C	S		
0 1 0 0	4	EOT	DC4	$	4	D	T		
0 1 0 1	5	WRU	ERR	%	5	E	U		
0 1 1 0	6	RU	SYNC	&	6	F	V		
0 1 1 1	7	BELL	LEM	'	7	G	W		
1 0 0 0	8	FE	S0	(	8	H	X		
1 0 0 1	9	HT/SK	S1	)	9	I	Y		
1 0 1 0	10 (A)	LF	S2	*	:	J	Z		
1 0 1 1	11 (B)	VT	S3	+	;	K	[		
1 1 0 0	12 (C)	FF	S4	,	<	L	\		ACK
1 1 0 1	13 (D)	CR	S5	—	=	M	]		①
1 1 1 0	14 (E)	SO	S6	.	>	N	↑		ESC
1 1 1 1	15 (F)	SI	S7	/	?	O	←		DEL

UNASSIGNED (6) — UNASSIGNED (7)

Table C-2. USASCII X3.4–1967

Bits b7 b6 b5 →					0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b4	b3	b2	b1	Row ↓ / Column →	0	1	2	3	4	5	6	7
0	0	0	0	0	NULL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYNC	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	y
1	0	0	1	9	HT	EM	)	9	I	Y	i	x
1	0	1	0	10 (A)	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11 (B)	VT	ESC	+	;	K	[	k	{
1	1	0	0	12 (C)	FF	FS	,	<	L	\	l	¦
1	1	0	1	13 (D)	CR	GS	—	=	M	]	m	}
1	1	1	0	14 (E)	SO	RS	.	>	N	^	n	~
1	1	1	1	15 (F)	SI	US	/	?	O	_	o	DEL

Table C-3. Extended Binary Coded Decimal Interchange Codes (EBCDIC)

b8 →	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
b7 →	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
b6 →	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
b5 →	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
b4 b3 b2 b1 COL → / ROW ↓	0	1	2	3	4	5	6	7	8	9	10 (A)	11 (B)	12 (C)	13 (D)	14 (E)	15 (F)
0 0 0 0 0	NUL 12-0-9-8-1	DLE 12-11-9-8-1			SP BLANK	& 12	- 11						PZ (Note 1) 12-0	MZ (Note 2) 11-0	\ 0-8-2	0 0
0 0 0 1 1	SOH 12-9-1	DC1 11-9-1					/ 0-1		a 12-0-1	j 12-11-1			A 12-1	J 11-1		1 1
0 0 1 0 2	STX 12-9-2	DC2 11-9-2		SYN 9-2					b 12-0-2	k 12-11-2	s 11-0-2		B 12-2	K 11-2	S 0-2	2 2
0 0 1 1 3	ETX 12-9-3	DC3 11-9-3							c 12-0-3	l 12-11-3	t 11-0-3		C 12-3	L 11-3	T 0-3	3 3
0 1 0 0 4									d 12-0-4	m 12-11-4	u 11-0-4		D 12-4	M 11-4	U 0-4	4 4
0 1 0 1 5	HT 12-9-5	NL 11-9-5	LF 0-9-5						e 12-0-5	n 12-11-5	v 11-0-5		E 12-5	N 11-5	V 0-5	5 5
0 1 1 0 6		BS 11-9-6	ETB 0-9-6						f 12-0-6	o 12-11-6	w 11-0-6		F 12-6	O 11-6	W 0-6	6 6
0 1 1 1 7	DEL 12-9-7		ESC 0-9-7	EOT 9-7					g 12-0-7	p 12-11-7	x 11-0-7		G 12-7	P 11-7	X 0-7	7 7
1 0 0 0 8		CAN 11-9-8							h 12-0-8	q 12-11-8	y 11-0-8		H 12-8	Q 11-8	Y 0-8	8 8
1 0 0 1 9		EM 11-9-8-1							i 12-0-9	r 12-11-9	z 11-0-9		I 12-9	R 11-9	Z 0-9	9 9
1 0 1 0 10 (A)		EM 11-9-8-2			[12-8-2	] 11-8-2	| 12-11	: 8-2								
1 0 1 1 11 (B)	VT 12-9-8-3				. (period) 12-8-3	$ 11-8-3	, (comma) 0-8-3	# 8-3								
1 1 0 0 12 (C)	FF 12-9-8-4	FS 11-9-8-4		DC4 9-8-4	< 12-8-4	* 11-8-4	% 0-8-4	@ 8-4								
1 1 0 1 13 (D)	CR 12-9-8-5	GS 11-9-8-5	ENQ 0-9-8-5	NAK 9-8-5	(12-8-5	) 11-8-5	_ (underscore) 0-8-5	' (apostrophe) 8-5								
1 1 1 0 14 (E)	SO 12-9-8-6	RS 11-9-8-6	ACK 0-9-8-6		+ 12-8-6	; 11-8-6	> 0-8-6	= 8-6								
1 1 1 1 15 (F)	SI 12-9-8-7	US 11-9-8-7	BEL 0-9-8-7	SUB 9-8-7	! 12-8-7	¬ 11-8-7	? 0-8-7	" 8-7								

*INTERNAL COLLATING SEQUENCE = b_8 b_7 b_6 b_5 / b_4 b_3 b_2 b_1

0 0 0 0 / 0 0 0 0 TO 1 1 1 1 / 1 1 1 1

NOTE 1: Graphics may vary on drum printers. Possibilities include + and {

NOTE 2: Graphics may vary on drum printers. Possibilities include ! and {.

Description of EBCDIC Codes

The following are descriptions of selected EBCDIC codes in Table C–3.

Control and Special Codes

NUL — NULL - The all-zeros character that can serve to accomplish time fill and media fill.

SOH — START OF HEADING - A communication control character used at the beginning of a sequence of characters that constitutes a machine-sensible address or routing information. Such a sequence is referred to as the *heading*. An STX character terminates a heading.

STX — START OF TEXT - A communication control character that precedes a sequence of characters to be treated as an entity and entirely transmitted through to the ultimate destination. Such a sequence is referred to as *text*. STX terminates a sequence of characters started by SOH.

ETX — END OF TEXT - A communication control character used to terminate a sequence of characters started with STX and transmitted as an entity.

HT — HORIZONTAL TABULATION - A format effector that controls the movement of the printing position to the next in a series of predetermined positions along the printing line. (Applicable also to display devices and the SKIP function on punched cards.)

DEL — DELETE - This character is used primarily to erase or obliterate extraneous or unwanted characters in perforated tape. (In the strict sense, DEL is not a control character.)

VT — VERTICAL TABULATION - A format effector that controls the movement of the printing position to the next in a series of predetermined printing lines. (Applicable also to display devices.)

FF — FORM FEED - A format effector that controls the movement of the printing position to the first predetermined printing line on the next form or page. (Applicable also to display devices.)

CR — CARRIAGE RETURN - A format effector that controls the movement of the printing position to the first printing position on the same printing line. (Applicable also to display devices.)

SO — SHIFT OUT - A control character indicating that the code combinations that follow are to be interpreted as outside of the character set of the standard code table until a shift in character is reached.

SI — SHIFT IN - A control character indicating that the code combinations that follow are to be interpreted according to the standard code table.

DLE — DATA LINK ESCAPE - A communication control character that changes the meaning of a limited number of contiguously following characters. It is used exclusively to provide supplementary controls in data communication networks.

DC1
DC2
DC3
DC4 — DEVICE CONTROLS - Characters for the control of ancillary devices associated with data processing or telecommunication systems, especially switching devices *on* or *off*. (If a single *stop* control is required to interrupt or turn off ancillary devices, DC4 is the preferred assignment.)

NL — NEW LINE - A format effector that causes both carriage return and line feed.

BS — BACKSPACE - A format effector that controls the movement of the printing position one printing space backward on the same printing line. (Applicable also to display devices.)

CAN — CANCEL - A control character used to indicate that the data with which it is sent is in error or is to be disregarded.

EM — END OF MEDIUM - A control character associated with the sent data that can be used to identify the physical end of the medium, or end of the used or wanted, portion of information recorded on a medium. (The position of this character does not necessarily correspond to the physical end of the medium.)

FS — FILE SEPARATOR
GS — GROUP SEPARATOR
RS — RECORD SEPARATOR
US — UNIT SEPARATOR - These information separators may be used within data in operational fashion, except that their hierarchical relationship is the following: FS is the most inclusive, then GS, then RS, and US is least inclusive. (The content and length of a file, group, record, or unit are not specified.)

LF — LINE FEED - A format effector that controls the movement of the printing position to the next printing line. (Applicable also to display devices.)

ETB — END OF TRANSMISSION BLOCK - A communication control character used to control the end of a block of data for communication purposes. ETB is used for blocking data where the block structure is not necessarily related to the processing format.

ESC ESCAPE - A control character intended to provide code extension (supplementary characters) in general information interchange. The escape character itself is a prefix affecting the interpretation of a limited number of contiguously following characters.

ENQ ENQUIRY - A communication control character used in data communication systems as a request for a response from a remote station. It can be used as a *who are you* (WRU) request to obtain identification. It can also be used to obtain station status, or both.

ACK ACKNOWLEDGE - A communication control character transmitted by a receiver as an affirmative response to a sender.

BEL BELL - A character to use for calling human attention. It can control alarm or attention devices.

SYN SYNCHRONOUS IDLE - A communication control character used by a synchronous transmission system in the absence of any other character to provide a signal from which synchronism can be achieved or retained.

EOT END OF TRANSMISSION - A communication control character used to indicate the conclusion of a transmission, which might have contained one or more texts and any associated headings.

NAK NEGATIVE ACKNOWLEDGE - A communication control character transmitted by a receiver as a negative response to a sender.

SUB SUBSTITUTE - A character that can be substituted for a character that is determined to be invalid or in error.

SP SPACE - A normally non-printing graphic character used to separate words. It is also a format effector that controls the movement of the printing position, one printing position forward. (Applicable also to display devices.)

Other Codes

PZ PLUS ZERO - Code 0100 1110 is never obtained from a BCL plus sign; plus zero (PZ) code 1100 0000 prints as a plus sign. Choice of a graphic for PZ can vary from system to system.

MZ MINUS ZERO - Minus zero (MZ) prints as an !. The choice of styling (! or |) for graphic code 0100 1111 (MZ) can also vary from system to system.

Table C-4. BCL Language Codes

BITS b_6					0	0	1	1
BITS b_5					0	1	0	1
b_4	b_3	b_2	b_1	Row \ Column	0	1	2	3
0	0	0	0	0	? ANY OTHER	SPACE	— 11	& 12
0	0	0	1	1	1 1	/ 0-1	J 11-1	A 12-1
0	0	1	0	2	2 2	S 0-2	K 11-2	B 12-2
0	0	1	1	3	3 3	T 0-3	L 11-3	C 12-3
0	1	0	0	4	4 4	U 0-4	M 11-4	D 12-4
0	1	0	1	5	5 5	V 0-5	N 11-5	E 12-5
0	1	1	0	6	6 6	W 0-6	O 11-6	F 12-6
0	1	1	1	7	7 7	X 0-7	P 11-7	G 12-7
1	0	0	0	8	8 8	Y 0-8	Q 11-8	H 12-8
1	0	0	1	9	9 9	Z 0-9	R 11-9	I 12-9
1	0	1	0	10 (A)	0	≠ 0-8-2	x 11-0	+ 12-0
1	0	1	1	11 (B)	# 8-3	, 0-8-3	$ 11-8-3	. 12-8-3
1	1	0	0	12 (C)	@ 8-4	% 0-8-4	* 11-8-4	[12-8-4
1	1	0	1	13 (D)	: 8-5	= 0-8-5	) 11-8-5	(12-8-5
1	1	1	0	14 (E)	> 8-6	] 0-8-6	; 11-8-6	< 12-8-6
1	1	1	1	15 (F)	≥ 8-7	|| 0-8-7	¬ 11-8-7	← 12-8-7

Appendix D
Debugging Aids

This appendix describes aids available to help in debugging V Series ANSI-74 COBOL programs.

Object Program Memory Layout

With the aid of the dollar option MAP, certain data and code blocks can be identified in an ANSI-74 COBOL object program. If present, these areas appear in the order of the following paragraphs:

Reserved Memory

Locations 0 through 99 of every program are called the program's Reserved Memory. The format of this area is shown in the following table:

Table D-1. Reserved Memory Layout

Location	Contents
0	Switch 8
1–7	Switches 1 through 7 respectively, one digit each
8–15	IX1
16–23	IX2
24–31	IX3
32–39	Undefined
40–45	Stack Pointer (Top Stack)
46–47	Undefined
48–63	Edit Table
64–99	Undefined

Segment Dictionary

This area is of variable length and begins at location 100. It includes all segment dictionary entries for the program.

Resident Segment

This area is the resident segment. It contains the following:

- Constants required by the resident code block
- Working-storage data items
- Label, FIB (File Information Block), and record areas for all files including, any relative and indexed file Key files
- All resident segment code

Overlayed Segments

This area contains all overlayable segment code and constants.

High Data

This area contains all the buffers for all files in the program, including any Key files. This section is present if any files are declared.

Stack

The last item in the object program is the stack. Minimum stack size is 320 digits for all programs.

Use a STACK SIZE clause in the Object-Computer paragraph to increase or decrease this default stack size.

Object Code File

Unreferenced data blocks are optimized out of the final code file produced by the COBOL compiler. Therefore, they are not part of the final code file and do not appear in any MAP information provided by the compiler.

The dollar option MAP is available to assist the user in locating items in the object code file. Refer to the explanation of this dollar option in Section 9 "Compiler Operation" of Volume 1 of this Manual and to Example E-6 "Output Produced by Dollar Option MAP" in Appendix E.

Debugging

To assist in error detection, COBOL provides the facility to monitor during program execution the transfers of control to user-selected procedures and values of user-selected data items.

The user statements required to accomplish this monitoring are included in the source program and can be compiled or not according to the presence or absence of the WITH DEBUGGING MODE clause in the source program. In addition, you can control execution of debugging code at run time by switch SW6. These features are described in the following paragraphs.

The features of the COBOL language that support the Debug module are the following:

- A special register, DEBUG-ITEM. This data item provides information on the conditions that caused the execution of the debugging section. This special register is generated automatically by the COBOL compiler if WITH DEBUGGING MODE clause (see item 2, below) is specified in the source program. Only one DEBUG-ITEM is allocated per program (refer to "USE Statement" in Section 6 of Volume 1).
- A compile-time switch. This switch is the WITH DEBUGGING MODE clause, which can be included in the SOURCE-COMPUTER paragraph in the Environment Division. Its presence indicates that all debugging statements, lines (see the next paragraph), and sections are to be compiled; its absence indicates that these items are to be compiled as comments.
- Debugging lines. These are any lines with D in column 7. If the WITH DEBUGGING MODE clause is present in the program (the compile-time switch), they are compiled into the program. Otherwise, these lines are treated as comments (refer to "Indicator Area" in Section 1 of Volume 1 and to "Source-Computer Paragraph" in Section 4 of Volume 1). Once debugging lines have been compiled into the program (that is, the WITH DEBUGGING MODE clause was present at compile-time), they are always executed at run time. Their execution is not controlled by the object time switch SW6 (see the next paragraph).
- An object time switch, SW6. This switch controls the execution of the statements in the USE FOR DEBUGGING section (see the next paragraph). This switch cannot be set or reset within the program, because it is controlled outside the COBOL environment. SW6 is on when the WITH DEBUGGING MODE clause is specified in the source program; SW6 is off when this clause is not specified in the source program. If the switch is on, the debugging section is executed at run time; if the switch is off, all debugging statements, lines, and sections described in the Procedure Division under the USE FOR DEBUGGING statement are ignored. You need not recompile the source program to provide or take away this facility.

  Note that the object time switch has no effect on the compilation of the source program, nor does it effect the execution of any debugging statements, lines, or sections compiled into the program.
- A USE FOR DEBUGGING statement. This statement identifies the user items that are to be monitored by the associated debugging section. The items that can be monitored are identifiers (data items), files, and procedures. For complete information on how to monitor these items and what information is then available, refer to "USE Statement" in Section 6 of Volume 1.

Appendix E
Program Examples

This appendix contains examples designed for reference or educational purposes. Presenting simple programs or portions of a program can relate certain elements of the COBOL language so that you can more easily perceive the entire spectrum of the rules and usage of the COBOL syntax.

Example E–1 shows how to use the STRING and UNSTRING statements.

```
010000    IDENTIFICATION DIVISION.
010100    PROGRAM-ID.  EX-STRING-UNSTRING.
010200 *  **  DESIGNED TO ILLUSTRATE STRING AND UNSTRING   ***
010300    ENVIRONMENT DIVISION.
010400    CONFIGURATION SECTION.
010500    SOURCE-COMPUTER.    V-300.
010600    OBJECT-COMPUTER.    V-300.
010700    DATA DIVISION.
010800    WORKING-STORAGE SECTION.
010900    77  START-END-POSITION      PIC 99 COMP.
011000    77  COUNTER-ITEM            PIC 99 COMP.
011100    77  DUMMY-ITEM              PIC X(30).
011200    01  NUMBER-ITEMS.
011300        05  ONE                 PIC 9.
011400        05  TWO                 PIC 9.
011500        05  THREE               PIC 9.
011600        05  FIVE                PIC 9         VALUE 5.
011700        05  SIX                 PIC 999       VALUE 666.
011800        05  NINE                PIC 9(6).
011900    01  OPERATION-ITEMS.
012000        05  PLUS-SIGN           PIC X.
012100        05  EQUAL-SIGN          PIC X.
012200    01  PUNCTUATION-ITEMS.
012300        05  PERIOD              PIC X.
012400        05  QUOTE-MARK          PIC X.
012500        05  QUESTION-MARK       PIC X.
012600        05  COMMA-MARK          PIC X.
012700    01  WORD-ITEMS.
012800        05  PLUS-WORD           PIC X(8).
012900        05  MINUS-WORD          PIC X(8).
013000        05  EQUALS-WORD         PIC X(8).
013100        05  TIMES-WORD          PIC X(8).
013200    PROCEDURE DIVISION.
013300    FIRST-PARAGRAPH.
```

Example E-1. Use of STRING and UNSTRING (cont.)

```
013400        MOVE "  123EQUALSMINUS" TO DUMMY-ITEM.
013500        MOVE 1 TO START-END-POSITION.
013600 *    GET PAST LEADING SPACES
013700        INSPECT DUMMY-ITEM
013800            TALLYING START-END-POSITION FOR LEADING SPACES.
013900 *    NOW DO SIMPLE UNSTRING
014000        UNSTRING DUMMY-ITEM INTO
014100            ONE FOR 1,
014200            TWO FOR 1,
014300            THREE FOR 1,
014400            EQUALS-WORD FOR 6,
014500            MINUS-WORD FOR 5
014600            WITH POINTER START-END-POSITION.
014700 *  VERY ELEMENTARY STRING
014800        MOVE SPACES TO DUMMY-ITEM.
014900        STRING
015000            THREE FOR 1,
015100            SPACE FOR 1,
015200            MINUS-WORD FOR 5,
015300            SPACE FOR 1,
015400            TWO FOR ONE,
015500            SPACE FOR 1,
015600            EQUALS-WORD FOR 6,
015700            SPACE FOR 1,
015800            ONE DELIMITED BY SIZE
015900            INTO DUMMY-ITEM.
016000        DISPLAY DUMMY-ITEM.
016100 *    STRINGING DIFFERENT ITEMS TOGETHER
016200        MOVE SPACES TO DUMMY-ITEM.
016300        MOVE 1 TO START-END-POSITION.
016400        STRING
016500            SIX FOR ONE,
016600            " PLUS ", FIVE, SPACE DELIMITED BY SIZE,
016700            EQUALS-WORD DELIMITED BY SPACE,
016800            SPACE, ONE, ONE DELIMITED BY SIZE
016900            INTO DUMMY-ITEM
017000            WITH POINTER START-END-POSITION.
017100        DISPLAY DUMMY-ITEM.
017200 *    STRINGING ONTO THE END OF THIS ITEM.
017300        STRING
017400            ". RIGHT?" DELIMITED BY SIZE
017500            INTO DUMMY-ITEM
017600            WITH POINTER START-END-POSITION.
017700        DISPLAY DUMMY-ITEM.
017800 *    UNSTRING DIFFERENT ITEM.
017900        MOVE ZERO TO COUNTER-ITEM, START-END-POSITION.
018000        MOVE "+,=""?""99999,PLUS,TIMES,EXTRA"
018100            TO DUMMY-ITEM.
018200        UNSTRING DUMMY-ITEM
018300            DELIMITED BY ","
```

Example E-1. Use of STRING and UNSTRING (cont.)

```
018400           OR QUOTES
018500           INTO
018600           PLUS-SIGN, DELIMITER IN COMMA-MARK,
018700           EQUAL-SIGN.
018800           QUESTION-MARK, DELIMITER IN QUOTE-MARK,
018900           NINE, COUNT IN COUNTER-ITEM,
019000           PLUS-WORD,
019100           TIMES-WORD,
019200           TALLYING IN START-END-POSITION;
019300           ON OVERFLOW
019400               DISPLAY "WE FOUND MORE THAN"
019500               START-END-POSITION "ITEMS TO UNSTRING".
019600       DISPLAY "WE FOUND THE DELIMITERS"
019700           COMMA-MARK "AND" QUOTE-MARK
019800           "AND A GROUP OF"
019900           COUNTER-ITEM "NINES".
020000       STOP RUN.
```

Example E-1. Use of STRING and UNSTRING

Example E–2 shows how to create a USE procedure handle a magnetic tape parity error.

```
010000    IDENTIFICATION DIVISION.
010100    PROGRAM-ID.  TAPE-ERRORS.
010200  * ** TAPE RECORDS WHICH CANNOT BE READ BECAUSE OF PARITY ***
010300  * ** ERRORS ARE WRITTEN OUT TO A PRINTER FILE           ***
010400    ENVIRONMENT DIVISION.
010500    CONFIGURATION SECTION.
010600    SOURCE-COMPUTER.   V-300.
010700    OBJECT-COMPUTER.   V-300.
010800    INPUT-OUTPUT SECTION.
010900    FILE-CONTROL.
011000        SELECT TAPEIN  ASSIGN TO TAPE 9 CHANNEL GCR.
011100        SELECT TAPOUT  ASSIGN TO TAPE 9 CHANNEL.
011200        SELECT PERROR  ASSIGN TO PRINTER WITH BACKUP ON DISKPACK.
011300    DATA DIVISION.
011400    FILE SECTION.
011500    FD  TAPEIN
011600        LABEL RECORDS ARE OMITTED
011700        VALUE OF FILENAME IS "INPUT".
011800    01  IN-REC                 PIC  X(100).
011900    FD  TAPOUT
012000        LABEL RECORDS ARE STANDARD
012100        VALUE OF FILENAME IS "OUTPUT".
012200    01  OUT-REC                PIC X(100).
012300    FD  PERROR.
012400    01  PRINT-REC              PIC X(132).
012500    WORKING-STORAGE SECTION.
012600    77  I                      PIC 99.
012700    PROCEDURE DIVISION.
012800    DECLARATIVES.
012900    PARITY-ERROR                SECTION.
013000        USE AFTER ERROR PROCEDURE ON TAPEIN.
013100    ERROR-ROUTINE.
013200        MOVE 1 TO I.
013300        MOVE IN-REC TO PRINT-REC.
013400        WRITE PRINT-REC.
013500    END DECLARATIVES.
013600    MAIN-PROCESSING             SECTION.
013700    OPENERS.
013800        OPEN INPUT TAPEIN  OUTPUT TAPOUT PERROR.
013900    PROCESSING.
014000        READ TAPEIN  AT END GO TO WRAP-UP.
014100        IF I = 1  MOVE 0 TO I  GO TO PROCESSING.
014200        MOVE IN-REC TO OUT-REC.
014300        WRITE OUT-REC.
014400        GO TO PROCESSING.
014500    WRAP-UP.
014600        CLOSE TAPEIN PURGE.
014700        CLOSE TAPOUT LOCK.
014800        CLOSE PERROR.
014900        STOP RUN.
```

Example E-2. USE Procedure to Handle a Parity Error

Example E–3 shows how to update an indexed sequential file.

```
010000    IDENTIFICATION DIVISION.
010100    PROGRAM-ID.  IX-UPDATE.
010200 *  ****  DEMONSTRATES UPDATING AN INDEXED SEQUENTIAL FILE  *****
010300    ENVIRONMENT DIVISION.
010400    CONFIGURATION SECTION.
010500    SOURCE-COMPUTER.    V-300.
010600    OBJECT-COMPUTER.    V-300.
010700    INPUT-OUTPUT SECTION.
010800    FILE-CONTROL.
010900        SELECT INTRAN  ASSIGN TO DISKPACK.
011000        SELECT MASTFL  ASSIGN TO DISKPACK.
011100            ORGANIZATION IS INDEXED   ACCESS MODE IS DYNAMIC
011200            RECORD KEY IS MS-FTR-NO
011300            ALTERNATE RECORD KEY IS MS-SOFT-ID      WITH DUPLICATES
011400            ALTERNATE RECORD KEY IS MS-SOFT-VERSION WITH DUPLICATES
011500            FILE STATUS IS MS-FILE-STATUS.
011600    DATA DIVISION.
011700    FILE SECTION.
011800    FD  INTRAN
011900        VALUE OF AREAS IS 20
012000        VALUE OF AREASIZE IS 1800
012100        VALUE OF FAMILYNAME IS "FTRMST"
012200        BLOCK CONTAINS 5 RECORDS.
012300    01  TRANS.
012400        03  TRANS-CODE           PIC 99.
012500            88  CHANGE-CODE      VALUE 01.
012600            88  DELETE-CODE      VALUE 02.
012700            88  ADD-CODE         VALUE 03.
012800        03  TX-FTR-NO.
012900            05  TX-FTR-REGN      PIC 999         COMP.
013000            05  TX-FTR-DIST      PIC 999         COMP.
013100            05  TX-FTR-BR-ID     PIC X(6).
013200        03  TX-SOFT-ID           PIC X(6).
013300        03  TX-SOFT-VERSN        PIC 99          COMP.
013400        03  TX-CONC-DESC         PIC X(80).
013500    FD  MASTFL
013600        VALUE OF AREAS IS 20
013700        VALUE OF AREASIZE IS 1800
013800        VALUE OF FAMILYNAME IS "FTRMST"
013900        BLOCK CONTAINS 5 RECORDS.
014000    01  MAST-REC.
014100        03  MS-FTR-NO.
014200            05  MS-FTR-REGN      PIC 999         COMP.
014300            05  MS-FTR-DIST      PIC 999         COMP.
014400            05  MS-FTR-BR-ID     PIC X(6).
014500        03  MS-SOFT-ID           PIC X(6).
014600        03  MS-SOFT-VERSION.
014700            05  MS-SOFT-VERSN    PIC 99          COMP.
```

Example E-3. Programmatic Handling of an Indexed File (cont.)

```
014800       03  MS-CONC-DESC         PIC X(80).
014900   WORKING-STORAGE SECTION.
015000   01  MISC.
015100       03  INDICATORS.
015200           05  MS-FILE-STATUS.
015300               07  MS-FILE-ST-1 PIC X.
015400               07  MS-FILE-ST-2 PIC X.
015500           05  MS-VALUES REDEFINES MS-FILE-STATUS PIC XX.
015600               88  SUCCESSFUL-STATUS   VALUE "00".
015700               88  EOF-STATUS          VALUE "10".
015800               88  DUP-KEY-STATUS      VALUE "22".
015900               88  NOT-FOUND-STATUS    VALUE "23".
016000           05  TRANS-EOF           PIC 99    COMP  VALUE ZERO.
016100   PROCEDURE DIVISION.
016200   PAR-1.
016300       PERFORM OPEN-FILES.
016400       PERFORM READ-TRANSACTION.
016500       PERFORM UPDATE-MASTER-FILE
016600           UNTIL TRANS-EOF = 1.
016700       PERFORM CLOSE-FILES.
016800       STOP RUN.
016900   READ-TRANSACTION.
017000       IF TRANS-EOF = ZERO
017100          READ INTRAN AT END
017200              MOVE 1 TO TRANS-EOF.
017300   UPDATE-MASTER-FILE.
017400       MOVE TX-FTR-NO TO MS-FTR-NO.
017500       IF NOT  EOF-STATUS
017600           READ MASTFL
017700               KEY IS MS-FTR-NO
017800               INVALID KEY
017900                   PERFORM EVALUATE-FILE-STATUS.
018000       IF SUCCESSFUL-STATUS
018100           IF  CHANGE-CODE    PERFORM CHANGE-IT
018200           ELSE
018300           IF DELETE-CODE     PERFORM DELETE-IT
018400           ELSE               PERFORM ERROR-TRANS
018500       ELSE
018600       IF (EOF-STATUS OR NOT-FOUND-STATUS)
018700           IF ADD-CODE        PERFORM ADD-IT
018800           ELSE               PERFORM ERROR-TRANS.
018900       PERFORM READ-TRANSACTION.
019000   OPEN-FILES.
019100       OPEN INPUT INTRAN.
019200       OPEN I-O MASTFL.
019300   CLOSE-FILES.
019400       CLOSE INTRAN.
019500       CLOSE MASTFL WITH DISMISS.
019600   EVALUATE-FILE-STATUS.
```

Example E-3. Programmatic Handling of an Indexed File (cont.)

```
019700         IF NOT EOF-STATUS OR NOT NOT-FOUND-STATUS
019800              DISPLAY "UNSUCCESSFUL READ " MS-FILE-STATUS
019900              CALL SYSTEM DUMP
020000              STOP RUN.
020100     CHANGE-IT.
020200     DELETE-IT.
020300     ADD-IT.
020400     ERROR-TRANS.
```

Example E-3. Programmatic Handling of an Indexed File

Example E–4 shows how to create ICMs.

```
010000 $  SET LIST LIST$ TITLE "EXAMPLE OF HOW TO CREATE ICMS"          1 EDITOR  DOLLAR
010100 * ?COMPILE MODULE1 COBOL LIB                                     2 EDITOR
010200 * ?DATA CARD                                                     3 EDITOR
010300 * ** THE FOLLOWING NAMES THE ICM FILE AND DIRECTS IT TO PACK ***  4 EDITOR
010400 $  SET ICM "SYSTEM/ICMEX1" PACK                                  5 EDITOR  DOLLAR
010500 $  RESET MAIN                                                    6 EDITOR  DOLLAR
010600   IDENTIFICATION DIVISION.                                       7 EDITOR
010700 * ** FOLLOWING IS PROGRAM-NAME USED IN CALL MODULE STATEMENT ***  8 EDITOR
010800   PROGRAM-ID.  EXMPL4.                                           9 EDITOR
010900   AUTHOR.  UNISYS CORPORATION.                                  10 EDITOR
011000   DATE-WRITTEN.  1989.                                          11 EDITOR
011100   DATE-COMPILED.  1989 MAY 18    01:12                          12 EDITOR
011110   SECURITY. 8395:CEX4 ON DOCPAK.                                13 EDITOR
011200   ENVIRONMENT DIVISION.                                         14 EDITOR
011300   CONFIGURATION SECTION.                                        15 EDITOR
011400   SOURCE-COMPUTER.  V-300.                                      16 EDITOR
011500   OBJECT-COMPUTER.  V-300.                                      17 EDITOR
011600   DATA DIVISION.                                                18 EDITOR
011700   WORKING-STORAGE SECTION.                                      19 EDITOR
011800 * ** THE FOLLOWING DATA BLOCK IS DECLARED AS A "COMMON" AREA ***  20 EDITOR
011900 * ** DATA CAN BE PASSED BETWEEN MODULES USING THIS 01-LEVEL ***   21 EDITOR
012000 *                                                                 22 EDITOR
012100   01  DATA-01 IS COMMON.                                         23 EDITOR
012200       03 FILLER    PIC X(20).                                    24 EDITOR
012300 *                                                                 25 EDITOR
012400   PROCEDURE DIVISION.                                           26 EDITOR
012500 *                                                                 27 EDITOR
012600   DISPLAYER SECTION.                                            28 EDITOR
012700   PARG-1.                                                       29 EDITOR
012800       DISPLAY DATA-01.                                          30 EDITOR
012900   EXIT-DISPLAYER.                                               31 EDITOR
013000       EXIT MODULE.                                              32 EDITOR
013100 *                                                                 33 EDITOR
013200   CHANGER SECTION.                                              34 EDITOR
013300   PARG-2.                                                       35 EDITOR
013400       MOVE "NEW CONTENTS" TO DATA-01.                           36 EDITOR
013500   EXIT-CHANGER.                                                 37 EDITOR
013600       EXIT MODULE.                                              38 EDITOR


          38 RECORDS PROCESSED
          NO WARNINGS
          NO CORRECTIONS
          NO ERRORS
          NO TOTAL DIAGNOSTICS LISTED
```

Example E-4. Binding Example—Creating ICMs

Example E–5 shows how to call ICMs.

```
010000 $  SET LIST LIST$ TITLE "MAIN MODULE EXAMPLE"                    1 EDITOR  DOLLAR
010200 * ?COMPILE CALLER COBOL LIB                                      2 EDITOR
010300 * ?DATA CARD                                                     3 EDITOR
010400   IDENTIFICATION DIVISION.                                       4 EDITOR
010500   PROGRAM-ID.  MAIN01.                                           5 EDITOR
010600   AUTHOR.  UNISYS CORPORATION.                                   6 EDITOR
010700   DATE-WRITTEN   1989.                                           7 EDITOR
010800   DATE-COMPILED.  1989 MAY 18    01:12                           8 EDITOR
010810   SECURITY.  8395:CEX5 ON DOCPAK.                                9 EDITOR
010900   ENVIRONMENT DIVISION.                                         10 EDITOR
011000   CONFIGURATION SECTION.                                        11 EDITOR
011100   SOURCE-COMPUTER.  V-300.                                      12 EDITOR
011200   OBJECT-COMPUTER.  V-300.                                      13 EDITOR
011300   DATA DIVISION.                                                14 EDITOR
011400   WORKING-STORAGE SECTION.                                      15 EDITOR
011500 *                                                               16 EDITOR
011600 * ** THIS FOLLOWING IS THE COMMON AREA ***                      17 EDITOR
011700 * ** USED TO PASS DATA BETWEEN PROGRAM PARTS ***                18 EDITOR
011800 *                                                               19 EDITOR
011900 * 01  DATA-01 IS COMMON.                                        20 EDITOR
012000       03 FILLER         PIC X(20).                             21 EDITOR
012100 *                                                               22 EDITOR
012200   01  DATA-02           PIX X(10).                             23 EDITOR
012300   PROCEDURE DIVISION.                                           24 EDITOR
012400   MAINLINE    SECTION.                                          25 EDITOR
012500   PARG-1.                                                       26 EDITOR
012600       MOVE SPACES TO DATA-02.                                   27 EDITOR
012700       MOVE "ORIGINAL CONTENTS" TO DATA-01.                      28 EDITOR
012710 * **  TO INVOKE A SECTION OF THE ICM FILE CREATED BY EXAMPLE 4 *** 29 EDITOR
012720 * **       CALL <SECTION-NAME> OF <PROGRAM-ID> ***              30 EDITOR
012730 * **       FROM <ICM-NAME> ON <PACK-NAME> ***                   31 EDITOR
012740 * ** NOTE: ICM MUST BE PRESENT AT COMPILE TIME                  32 EDITOR
012800       CALL MODULE "DISPLAYER" OF "EXMPL4"                       33 EDITOR
012900           FROM "ICMEX1" ON "SYSTEM".                            34 EDITOR
013000       CALL MODULE "CHANGER" OF "EXMPL4"                         35 EDITOR
013100           FROM "ICMEX1" ON "SYSTEM".                            36 EDITOR
013200       CALL MODULE "DISPLAYER" OF "EXMPL4"                       37 EDITOR
013300           FROM "ICMEX1" ON "SYSTEM".                            38 EDITOR
013400       STOP RUN.                                                 39 EDITOR

  OVERALL MEMORY LAYOUT:

  -----REGION-----    BASE    SIZE
  RESERVED MEMORY        0     140
  SEG DICTIONARY       140      96
  RESIDENT SEG         236     492
  STACK                728    1272
  TOTAL PROGRAM          0    2000
```

Example E-5. Binding Example—Calling ICMs

Example E–6 shows the output produced by dollar option MAP.

```
010000 * ?CMP MAPTST COBOL LIB                                          1 EDITOR
010100 * ?DATA CARD                                                     2 EDITOR
010200 $ SET LIST LIST$ MAP                                             3 EDITOR DOLLAR
010300   IDENTIFICATION DIVISION.                                       4 EDITOR
010400   PROGRAM-ID. MAPTST.                                            5 EDITOR
010500 * ***   DESIGNED TO SHOW $MAP OUTPUT    ****                     6 EDITOR
010600   ENVIRONMENT DIVISION.                                          7 EDITOR
010700   CONFIGURATION SECTION.                                         8 EDITOR
010800   SOURCE-COMPUTER.  V-300.                                       9 EDITOR
010900   OBJECT-COMPUTER.  V-300.                                      10 EDITOR
011000   INPUT-OUTPUT SECTION.                                         11 EDITOR
011100   FILE-CONTROL.                                                 12 EDITOR
011200       SELECT TSTIND ASSIGN TO DISK ORGANIZATION IS INDEXED      13 EDITOR
011300           RECORD KEY IS KEY1 ALTERNATE RECORD KEY IS KEY2       14 EDITOR
011400           FILE STATUS IS TST-STATUS.                            15 EDITOR
011500       SELECT TSTSEQ ASSIGN TO DISK RESERVE 5 AREAS.             16 EDITOR
011600   DATA DIVISION.                                                17 EDITOR
011700   FILE SECTION.                                                 18 EDITOR
011800   FD  TSTIND                                                    19 EDITOR   1     672
011900       BLOCK CONTAINS 9 RECORDS                                  20 EDITOR
012000       RECORD CONTAINS 80 CHARACTERS                             21 EDITOR
012100       VALUE OF FAMILYNAME IS "TOPACK"                           22 EDITOR
012200               FILENAME IS "INDFLE"                              23 EDITOR
012300               AREAS 10                                          24 EDITOR
012400               AREASIZE 90.                                      25 EDITOR
012500   01  INDEXED-RECORD.                                          26 EDITOR   1    5036
012600       05  KEY1                PIC X(10).                       27 EDITOR   1    5036
012700       05  KEY2                PIC X(10).                       28 EDITOR   1    5056
012800       05  THE-REST            PIC X(60).                       29 EDITOR   1    5076
012900   FD  TSTSEQ                                                    30 EDITOR   1    1976
013000       BLOCK CONTAINS 1 RECORDS                                  31 EDITOR
013100       RECORD CONTAINS 80 CHARACTERS                             32 EDITOR
013200       VALUE OF FILENAME IS "SEQFLE".                            33 EDITOR
013300   01  SEQUENTIAL-RECORD       PIC X(80).                       34 EDITOR   1    5204
013400   WORKING-STORAGE SECTION.                                      35 EDITOR
013500   77  FIRST-77-ITEM           PIC X.                           36 EDITOR   1    5366
013600   77  SECOND-77-ITEM          PIC X.                           37 EDITOR   1    5368
013700   01  TST-STATUS.                                              38 EDITOR   1    6012
013800       05  STATUS-1            PIC X.                           39 EDITOR   1    6012
013900       05  STATUS-2            PIC X.                           40 EDITOR   1    6014
014000 *  THE FOLLOWING IS AN UNREFERENCED DATA BLOCK                  41 EDITOR      000002
014100 *  NOTE THAT IT DOES NOT APPEAR IN MAP SUMMARY                  42 EDITOR      000002
0141200  01  UNREFERENCED-DATA.                                       43 EDITOR NOT REFERENCED
014300       05  ITEM-1              PIC 9.                           44 EDITOR NOT REFERENCED
014400       05  ITEM-2              PIC 9.                           45 EDITOR NOT REFERENCED
014500  PROCEDURE DIVISION.                                           46 EDITOR   1   53830
014600  1010-SECTION                 SECTION 1010.                    47 EDITOR   1   53830
014700  1010-PARA-1.                                                  48 EDITOR   1   53774
```

Example E-6. Output Produced by Dollar Option Map (cont.)

```
014800      OPEN OUTPUT TSTIND.                                    49 EDITOR   1   53774
014900  1040-SECTION                SECTION 1040.                  50 EDITOR   1   53822
015000  1040-PARA-1.                                               51 EDITOR   1   53744
015100      OPEN OUTPUT TSTSEQ.                                    52 EDITOR   1   53744
015200  1060-SECTION                SECTION 1060.                  53 EDITOR   2   53972
015300  1060-PARA-1.                                               54 EDITOR   2   53972
015400      MOVE ALL 1 TO INDEXED-RECORD.                          55 EDITOR   2   53972
                 1                                               FIRST MESSAGE
******     1) WARNING: NONSTANDARD USAGE WILL BE DISALLOWED IN FUTURE RELEASE
15500  2010-SECTION                SECTION 2010.                  56 EDITOR   2   53990
015600  2010-PARA-1.                                               57 EDITOR   1   53672
015700      WRITE INDEXED-RECORD  INVALID KEY STOP RUN.            58 EDITOR   1   53672
015800  2040-SECTION                SECTION 2040.                  59 EDITOR   1   53736
015900  2040-PARA-1.                                               60 EDITOR   1   53646
016000      MOVE ALL 2 TO SEQUENTIAL-RECORD.                       61 EDITOR   1   53646
                 1                              LAST MESSAGE AT 154
******     1) WARNING: NONSTANDARD USAGE WILL BE DISALLOWED IN FUTURE RELEASE
016100  2060-SECTION                SECTION 2060.                  62 EDITOR   3   54000
016200  2060-PARA-1.                                               63 EDITOR   3   54000
016300      WRITE SEQUENTIAL-RECORD  INVALID KEY STOP RUN.         64 EDITOR   3   54000
016400  3020-SECTION                SECTION 3020.                  65 EDITOR   3   54062
016500  3020-PARA-1.                                               66 EDITOR   1   53598
016600      CLOSE TSTIND DISMISS.                                  67 EDITOR   1   53598
016700  4040-SECTION                SECTION 4040.                  68 EDITOR   1   53638
016800  4040-PARA-1.                                               69 EDITOR   1   53568
016900      CLOSE TSTSEQ DISMISS.                                  70 EDITOR   1   53568
017000  7070-SECTION                SECTION 7070.                  71 EDITOR   4   54072
017100  7070-PARA-1.                                               72 EDITOR   4   54072
017200      CALL SYSTEM DUMP.                                      73 EDITOR   4   54072
017300  9191-SECTION                SECTION 9191.                  74 EDITOR   5   54140
017400  9191-PARA-1.                                               75 EDITOR   5   54140
017500      MOVE "X" TO SECOND-77-ITEM.                            76 EDITOR   5   54140
017600  9999-PARA-2.                                               77 EDITOR   5   54148
017700      STOP RUN.                                              78 EDITOR
```

```
DETAILS OF SEGMENT #1                  ***  { PASS }  ***

  BASE   SIZE   ----------BLOCK NAME----------
DATA BLOCKS
   512    400   MAPTST.TSTIND
   912    450          IND-KEY01-FILE-002
  1364    450          IND-KEY02-FILE-003
  1816    560          TSTSEQ
  2376      7*  IND.WRITE-BLOCK.CONST-POOL
  2384     19*      RETRIEVE-TABLE.CONST-POOL
  2404      8*      EXCEPTION-HANDLER.CONST-POOL
```

Example E-6. Output Produced by Dollar Option Map (cont.)

```
   2412     24*      OPEN-FILE-IO.CONST-POOL
   2436     24*      OPEN-FILE-OI.CONST-POOL
   2460      8*      UPDATE-DF-CTL-BLK.CONST-POOL
   2468      8*      INSERT.ACON-POOL
   2476     37*      IND-OPEN.CONST-POOL
   2516     40*      IND-OPEN.ACON-POOL
   2556      7   MAPTST.1010-SECTION.CONSTANTS
   2564      8          1010-SECTION.ADDRESS_CONSTANTS
   2573     14          2010-SECTION.ADDRESS_CONSTANTS
   2588      7          3020-SECTION.CONSTANTS
   2596      8          3020-SECTION.ADDRESS_CONSTANTS
   2604     44   BSR.ISSUE-SET-ATTR-BCT.ISSUE-SET-ATTR-BCT
   2648     12       GET-DFH-INFO.GET-DFH-INFO
   2660    108       RETURN-ERROR.RETURN-ERROR
   2768     12       MACHINE-SEARCH.MACHINE-SEARCH
   2780      8       MOVE-BUFF.MOVE-BUFF
   2788      8       MOVE-MOD2.MOVE-MOD2
   2796     52       CONVERT-KEY.CONVERT-KEY
   2848     52       KEY-COMPARE.KEY-COMPARE
   2900     12       DATE-TIME.DATE-TIME
   2912      8       ISSUE-BCT.ISSUE-BCT
   2920   1264   ERR-DATA
   4184    130   BSR.GLOBAL
   4316     24   IND.CONST-POOL
   4340     40   DFH-INFO
   4380     14   OPEN-INFO
   4390     46   IND.OPEN-CLOSE-DATA
   4444     20       FILE-DATA
   4464    562       IND-DATA
   5028    168   MAPTST.REC-001-TSTIND
   5196    168          REC-004-TSTSEQ
   5364      6          LEVEL-77'S
   5372    160          BCS-TABLE-01
   5532    480          IF-BLOCK-01
   6012      4          TST-STATUS
CODE BLOCKS
   6016    292   IND.GET-NEW-TABLE
   6308   2360       FIXUP-TABLE
   8668   1084       SEARCH-THRU-DUPS
   9572   1152   BSR.KEY-COMPARE
  10904   2536   IND.TRAVERSE-TREE
  13440    930   BSR.MACHINE-SEARCH
  14370    210       ZERO-DATA
  14580   1836   IND.SET-UP-NEXT-TABLE
  16416     98   BSR.HANDLER-RETURN-TABLE
  16514   1228   IND.OPEN-KEY-FILE
  17742    374       COMPARE-FD-FIELDS
  18116    254   BSR.GET-DFH-INFO
  18370    282       ISAM-OPEN-AVAILABLE
  18652    694   IND.SET-KF-ATTRIBUTES
```

Example E-6. Output Produced by Dollar Option Map (cont.)

```
 19346    178   BSR.ISSUE-SET-ATTR-BCT
 19524    622       MOVE-MOD2
 20146    652   IND.RIGHT-JUSTIFY-TABLE
 20798   1690       SPLIT-TABLE
 22488    848       SEARCH-TABLE
 23336   3310       RETRIEVE-TABLE
 26646    784       INITIALIZE-TREE
 27430    444   BSR.CONVERT-KEY
 27874    484   IND.REMOVE
 28358    614       WRITE.BLOCK
 28972    584   BSR.RETURN-ERROR
 29556    288       ZERO-BUFF
 29844    160       DATE-TIME
 30004    514       ISSUE-BCT
 30518   1050   IND.FLUSH-ALL-BUFFERS
 31568    622       RECOVER-WRITE
 32190   2620       INSERT
 34810    282   BSR.MOVE-BUFF
 35092   1866   IND.RETRIEVE-BLOCK
 36958   2388       UPDATE-DF-CTL-BLK
 38640   2388       OPEN-FIL-OI
 41028   3200       OPEN-FILE-IO
 44228   1802       OPEN-FILE-INPUT
 46030    574       EXCEPTION-HANDLER
 46604   1588       IND-OPEN
 48192   2560       IND-WRITE
 50752   2816       IND-CLOSE
 53568     30   MAPTST.4040-SECTION
 53598     48          3020-SECTION
 53646     26          2040-SECTION
 53672     72          2010-SECTION
 53744     30          1040-SECTION
 53774     56          1010-SECTION
 53830      8          RESIDENT-CODE-BLOCK-2


DETAILS OF SEGMENT # 2:    SECTION 010060

  BASE   SIZE   ----------BLOCK NAME----------
CODE BLOCKS
 53972     26          1060-SECTION



DETAILS OF SEGMENT # 3:    SECTION 020060

  BASE   SIZE   ----------BLOCK NAME----------
CODE BLOCKS
 54000     70          2060-SECTION
```

Example E-6. Output Produced by Dollar Option Map (cont.)

```
 DETAILS OF SEGMENT # 4:    SECTION 070070

   BASE   SIZE   ----------BLOCK NAME----------
 CODE BLOCKS
  54072     24          7070-SECTION


 DETAILS OF SEGMENT # 5:    SECTION 090191

   BASE   SIZE   ----------BLOCK NAME----------
 CODE BLOCKS
  54140     66          9191-SECTION


DETAILS OF HIGH DATA

DATA BLOCKS
  54208   4500          BUF_002_KEY03-TSTIND
  58708   4500          BUF_002_KEY02-TSTIND
  63208   4500          BUF_002-KEY01-TSTIND
  67708    800          BUF_004-TSTSEQ
  68508   1848          BUF_001-TSTIND
OVERALL MEMORY LAYOUT

-----REGION-----    BASE    SIZE
RESERVED MEMORY        0     140
SEG DICTIONARY       140     192
RESIDENT SEG         332   53640
OVERLAYED SEGS     53972     236
HIGH DATA          54208   16148
STACK              70356    1644
TOTAL PROGRAM          0   72000


SEGMENT LAYOUT:

---SEGMENT NAME-----  SEG   BASE  SIZE  SLACK  INTERCEPT CODE
                       1    332  53640  NONE       132
  SECTION 010060       2  53972     28   208         0
  SECTION 020060       3  54000     72   136         0
  SECTION 070070       4  54072     68    68        44
  SECTION 090191       5  54140     68     0         0
```

Example E-6. Output Produced by Dollar Option Map (cont.)

```
SEG  LO----------------------------SPATIAL RELATIONSHIPS-------------------------HI

  2                                                                               X

  3                                                                               X

  4                                                                                X

  5                                                                                X


              78 RECORDS PROCESSED
               2 WARNINGS
              NO CORRECTIONS
              NO ERRORS
               2 TOTAL DIAGNOSTICS LISTED
                   LAST DIAGNOSTIC AT RECORD NUMBER 160
```

Example E-6. Output Produced by Dollar Option Map

Example E–7 shows how to pass tape label information.

```
000100   IDENTIFICATION DIVISION.
000200   PROGRAM-ID. TAPE-LABELS.
000300 *
000400 *     This program reads and writes a tape file,
000500 *     passing label information from input to
000600 *     output.
000700 *
000800   ENVIRONMENT DIVISION.
000900   CONFIGURATION SECTION.
001000   SOURCE-COMPUTER. V-300.
001100   OBJECT-COMPUTER. V-300.
001200   INPUT-OUTPUT SECTION.
001300   FILE-CONTROL.
001400
001500       SELECT TAPEIN              ASSIGN TO TAPE.
001600       SELECT TAPEOT              ASSIGN TO TAPE.
001700
001800   I-O-CONTROL.
001900       SAME RECORD AREA FOR TAPEIN, TAPEOT.
002000
002100   DATA DIVISION.
002200   FILE SECTION.
002300
002400   FD  TAPEIN
002500       LABEL RECORDS ARE STANDARD
002600       VALUE OF LABELSIZE IS 100
002700       VALUE OF FILENAME IS "TRNTAP".
002800   01  INPT-RCD              PIC X(80).
002900
003000   FD  TAPEOT
003100       LABEL RECORD IS WS-LABEL
003200       VALUE OF LABELKIND IS STANDARD
003300       VALUE OF FILENAME IS "TRNTAP".
003400   01  OTPT-RCD              PIC X(80).
003500
003600   WORKING-STORAGE SECTION.
003700   01  WS-LABEL.
003800       05  STD-LABEL         PIC X(80).
003900       05  USR-AREA.
004000           10  POST-DT       PIC 9(6).
004100           10  USER-INFO     PIC X(14).
004200
004300   PROCEDURE DIVISION.
004400
004500   DECLARATIVES.
004600   OUTPUT-TAPE-BEGIN SECTION.
004700       USE AFTER STANDARD BEGINNING FILE LABEL
004800           PROCEDURE ON TAPEOT.
```

Example E-7. USE Procedure to Handle Tape Labels (cont.)

```
004900   OTPT-BGN-RTN.
005000       ACCEPT POST-DT FROM TODAYS-DATE.
005100   END DECLARATIVES.
005200
005300   MAIN SECTION.
005400   SQUARE-ONE.
005500       OPEN INPUT TAPEIN.
005600       CALL "INTRINSIC GETLABEL".
005700           USING TAPEIN GIVING WS-LABEL.
005800       OPEN OUTPUT TAPEOT.
005900
006000   PROCESS-LOOP.
006100       READ TAPEIN  AT END
006200           GO TO THATS-ALL-FOLKS.
006300       WRITE OTPT-RCD.
006400       GO TO PROCESS-LOOP.
006500
006600   THATS-ALL-FOLKS.
006700       CLOSE TAPEIN WITH LOCK.
006800       CLOSE TAPEOT WITH LOCK.
006900       STOP RUN.
```

Example E-7. USE Procedure to Handle Tape Labels

Appendix F
COBOL Conversion Aid

The COBOL74 Conversion Aid Program COFLTR facilitates the move from Unisys B 2000/B 3000/B 4000/V Series COBOL68 (Unisys COBOL68) to Unisys 1974 Corporate Standard COBOL (Unisys COBOL74).

General Description

COFLTR scans a syntactically correct COBOL source program and creates a new source program, a listing of the converted program and, optionally, a listing of the original program. If you attempt to process a COBOL program that is not syntactically correct through the Conversion Aid, the results are unpredictable.

The Conversion Aid provides for use of optional control records to select among input, output, and list options, to change the Object-Computer name, to convert switch references to data-name references, and to convert library files. An entire program is read and written. Any statements not changed or deleted are written as they appear.

Constructs can be flagged for any of the following reasons: they require manual change, they might produce different results after conversion than those produced before, or new rules of usage apply. All lines added, changed, deleted, or flagged have messages to the right on the listing.

Restrictions

A COBOL program that is input to the Conversion Aid must be able to be compiled syntax error free by the COBOLV or COBOLL compiler and must be blocked 1 through 10 or 20. Unisys recommends that you compile a program using COBOLV to test it before it is converted to COBOL74.

Because the Conversion Aid changes certain constructs, output from it is not acceptable as input to any of these compilers nor can it be re-input to the Conversion Aid and give correct results. A COBOL program should be run through the Conversion Aid one time only. If it must be rerun, you must use preconversion input.

To prevent a user from inadvertently recycling output from the Conversion Aid, the Conversion Aid inserts this initial comment record in the new source file:

```
*THIS SOURCE FILE HAS BEEN CONVERTED TO COBOL74.
```

This record halts subsequent execution of the Conversion Aid.

Control Statements

The following control statements are used to execute the COBOL Conversion Aid:

```
? EXECUTE COFLTR
? FILE file-name = file-id
? DATA COBCIF
& Option Control record
? END
```

File Equate (Label Equate) Information

Internal and external file names used by the Conversion Aid are identical and are as follows:

- COBCIF – This is COBOL source input from the card reader. It includes the control statements to run the program; hence it is a required file.
- COBDIF – This is COBOL source input from tape, disk, or diskpack. This file is not used if input is from cards.
- COBDOF – This is COBOL source output to tape, disk, diskpack, or punch. It is an optional file, not used unless the list option is SYNTAX.
- PRNTIF – Listing of COBOL source input assigned to the printer. It is an optional file, used only if the list option is LISTOLD.
- PRNTOF – Listing of COBOL source output and all messages assigned to the printer. It is a required file.

These five file names are the internal file names to use in the FILE statement. Refer to the *V Series Systems Operations Guide, Volume 2: System Commands*.

The following indicates the input character code set:

```
? DATA COBCIF indicates EBCDIC source language input.

? DATAB COBCIF indicates BCL source language input.
```

The output character code set is EBCDIC.

Use of the & Option Control Record

The purpose of the & option control record is to specify certain user options to the COBOL Conversion Aid Program. The & record specifies input option, output option, list option, resequence start and increment values if different from default values, and optionally the Object-Computer name, conversion of switch references to data-name references, and conversion of library files.

The & record is optional, and if not used, the program uses the following default values: DISKIN, DISKOUT, and LISTNEW. Resequence (RESEQ) is automatic. The default start value is 100, and the default increment is +10. The default Object-Computer name is that contained in the source program. The default treatment of references to switches (SW1...SW8) is flagging them as requiring manual change. The COPY default is nonconversion of library files.

More than one & record can be in the input stream; however, the last one overrides any preceding ones. Errors on all & records are flagged. Errors on the last & record halt program execution.

COBOL Compiler Control records ($ in column 7) can be present in the input stream. The beginning of the COBOL program is signaled by the first record with blank ( ), asterisk (*), or slash (/) in column 7.

The & record must contain an & in column 7. Columns 8 through 80 are available to the user and are freeform. One or more blanks separate the options. Punctuation marks must not be used. The options including RESEQ can appear in any order.

The presence of an & record nullifies the original default values of the input, output, and list options. Therefore, if the & record is used, it must contain one input option and one list option. Unless the list option is SYNTAX, the & record must also contain one output option.

The following are the input options:

- CARDIN – Specifies symbolic input is from the card reader.
- TAPEIN – Specifies symbolic input is from tape.
- DISKIN – Specifies symbolic input is from disk.
- PACKIN – Specifies symbolic input is from diskpack.

The following are the output options:

- CARDOUT – Specifies symbolic output is to cards.
- TAPEOUT – Specifies symbolic output is to tape.
- DISKOUT – Specifies symbolic output is to disk.
- PACKOUT – Specifies symbolic output is to diskpack.

The following are the list options:

- LISTNEW – Lists the symbolic as it appears after processing by the Conversion Aid.
- LISTOLD – Produces two separate listings: one of the original symbolic and one of the symbolic as it appears after being converted.
- LISTUN – Lists only those symbolics that could not be converted with their new sequence numbers and flag messages.

- SYNTAX – Does not create a new symbolic, but provides a projected listing of the symbolic after processing by the Conversion Aid. Listing is comparable to LISTNEW.

If SYNTAX and an output option are both specified on an & record, SYNTAX overrides the output option. A message to this effect is written on the listing, and program execution continues.

The following are the remaining & record options:

- RESEQ [nn] [+nn]
- NORESEQ
- OBJECT-COMPUTER name
- SWITCH
- COPY [NEWNAME]

RESEQ Option

Use of the word RESEQ is optional. If the user desires resequence values other than default values, he or she specifies RESEQ followed by the start value and increment value in that order. Each value field is allowed a length of six positions. Leading zeros can be omitted. One or more blanks separate the fields. The increment value must begin with a plus sign (+).

The user can specify start value without specifying increment value; he or she likewise can specify increment value without specifying start value. Either value not specified reverts to its default. Specifying RESEQ with no values has the same effect as omitting it.

NORESEQ Option

The option NORESEQ suppresses automatic resequencing. If both RESEQ and NORESEQ are specified on an & record, then NORESEQ overrides RESEQ. See "Sequence Numbers" in this appendix for a description of the operation of both RESEQ and NORESEQ and possible exception conditions.

OBJECT-COMPUTER Name Option

If the user wishes to change the OBJECT-COMPUTER model name as given in the source program or override the default name of V-300, he or she can specify an OBJECT-COMPUTER name on the & record. OBJECT-COMPUTER names as they should appear on the & record are V-300, V-400, and V-500.

Refer to "OBJECT-COMPUTER Paragraph" in this appendix for more information concerning OBJECT-COMPUTER names.

SWITCH Option

If switches (SW1...SW8) are used as data names and are set within the COBOL source program and not by a ?EXECUTE record, the user might want them converted to data names rather than flagged as invalid. To convert them, the user specifies the word SWITCH on the & record. Doing so causes the insertion of level 77 data names SW1-XX through SW8-XX (all PIC 9 COMP VALUE ZERO). All references to switch names SW1...SW8 are changed to SW1-XX...SW8-XX. Treatment of switches is discussed further under "Procedure Division" (see "Switches") in this appendix.

COPY Option

If the user wishes to convert library files that are accessed by the word COPY in the source program, he or she can specify the word COPY on the & record.

The library files might be present on disk or diskpack, depending on the syntax of the COPY statement in the source program. For example, "COPY FILEA" causes COFLTR to search for FILEA on disk and, if requested, generate a new library file to disk. "COPY FILEA ON PACKA" would cause COFLTR to search on pack PACKA for file FILEA and, if requested, generate a new library file to pack PACKA. If the generic name "PACK" is used, the search is done on the system resource diskpack.

A library file is included in a source listing following the COPY statement. Its records are identified by an *L* to the right of the statement. A library output file is created on the same media as the input library file if any output option that creates a new source file is specified. If NEWNAME is not specified to COFLTR, any output library file replaces the input library file. Conversion in the Data Division is discussed further under "COPY Statement" in this appendix. Several programs can COPY the same library file. However, the file can be converted only one time. Insert a comment message as the first record of a converted library file, preventing subsequent conversion.

If COPY is specified, and the user wants both the original library file and the converted library file to exist, he or she can specify NEWNAME. The original file retains the original name, but any output file as well as the listing generated by COFLTR reflect the new name.

The converted filename is the original name with the first character replaced with Q (Z if the original name starts with Q). An effect of using this option is that if the same library file is copied more than one time in a program, it is converted each time, and each subsequent time, the system issues a DUP LIB message. If NEWNAME is specified without COPY, it is ignored.

Examples of & Option Control Records

The following are some examples of & records:

```
&  CARDIN  TAPEOUT  LISTNEW
&  TAPEIN TAPEOUT LISTOLD RESEQ 1000 +100
&  LISTNEW  RESEQ 1000   DISKIN   DISKOUT
&  CARDIN SYNTAX V-300 COPY
&  DISKIN DISKOUT LISTNEW SWITCH
```

Error Conditions With & Option Control Records

The following conditions are errors that stop program execution:

- Presence of an unrecognized record preceding the COBOL source program. Only records that have & or $ in column 7 are recognized.
- The last & record does one of the following:
  - Contains a word not recognized as a valid option.
  - Omits an input or list option. Omits an output option and the list option is not SYNTAX.
  - Specifies RESEQ values that are not numeric or increments value with one or more blanks between + and a number.
  - Specifies CARDIN and a record having a blank, *, /, or $ in column 7 does not follow.
  - Specifies two or more input options.
  - Specifies two or more output options.
  - Specifies two or more list options.
  - Specifies two or more OBJECT-COMPUTER name options.
- Previously converted program. The following is the first record of the file:

```
*THIS SOURCE FILE HAS BEEN CONVERTED TO COBOL74
```

What the COBOL Conversion Aid Does

Changes that are processed by the Conversion Aid are presented in the following paragraphs according to the division to which they apply. Changes pertaining to more than one division are presented last.

Identification Division

The MONITOR statement is deleted.

The statement *PROGRAM-ID. COBOL74.* is inserted if the Program ID is not the first paragraph.

The program name is classified by Unisys COBOL74 as a user-defined word, and as such, it cannot be a reserved word. If the program name is a new reserved word, it is appended with XX. (New reserved words in other divisions are appended with -XX, as described under "New Reserved Words" in this appendix.) If the program name is enclosed in quotation marks, they are removed. If the program name consists of more than one word, all words after the first are deleted.

Entries following paragraph-names other than PROGRAM-ID in the Identification Division are comment entries, and as such, can be reserved words.

Optional paragraph names are checked for the correct order of presentation (AUTHOR, INSTALLATION, DATE-WRITTEN, DATE-COMPILED, SECURITY). An entry that is not in correct order is converted to comments by inserting an asterisk (*) in column 7.

The REMARKS paragraph is marked as comments by inserting an asterisk (*) in column 7.

The Identification Division header and all paragraph names are moved to begin in Area A (column 8) if they do not already begin there.

Extraneous words are removed from the division header.

Environment Division

In the Environment Division, section names and paragraph names are moved to begin in Area A (column 8) if they do not already begin there.

SELECT statements are moved to begin in Area B (column 12) if they do not already begin there.

Configuration Section

The Environment Division is required to contain the following section header and paragraphs:

```
CONFIGURATION SECTION.
SOURCE-COMPUTER. Computer-name.
OBJECT-COMPUTER. Computer-name.
```

If this section header or either paragraph is absent, it is inserted in proper order after the Environment Division header.

SOURCE-COMPUTER Paragraph

The SOURCE-COMPUTER name is regarded as a comment entry in both Unisys COBOL68 and Unisys COBOL74. An existing entry is not edited; however, an entry must be present, and it must be one word. If an entry consists of more than one word, all words after the first are deleted. If the Conversion Aid inserts the SOURCE-COMPUTER paragraph or only the computer name, the name used is V-300.

OBJECT-COMPUTER Paragraph

The OBJECT-COMPUTER name is regarded as a comment entry in Unisys COBOL68 except for the word V-300. Unisys COBOL 74 specifies that the computer name be one word that identifies the equipment configuration.

In the absence of an OBJECT-COMPUTER name entry on the & record, the name conversions are made as shown in Table F–1.

Table F-1. OBJECT-COMPUTER Name Conversions

Old Entry	New Entry
B-2500 (B2500)	V-300
B-2700 (B2700)	V-300
B-3500 (B3500)	V-300
B-3700 (B3700)	V-300
B-4700 (B4700)	V-300
B-2800 (B2800)	V-300
B-3800 (B3800)	V-300
B-4800 (B4800)	V-300
B-2900 (B2900)	V-300
B-3900 (B3900)	V-300
B-4900 (B4900)	V-300
Unrecognized name	V-300
Extended	V-300

If the Conversion Aid inserts the OBJECT-COMPUTER paragraph or only the computer name and the & record does not specify OBJECT-COMPUTER name, then the name used is V-300. See "Control Statement" in this appendix for specification of OBJECT-COMPUTER name on the & record.

The phrase WITH SUPERVISOR CONTROL is deleted. It is documentational only.

In the MEMORY SIZE integer clause in Unisys COBOL68, the units designator WORDS, CHARACTERS, or MODULES can be omitted, and the integer is assumed to be digits. Unisys COBOL74 requires the units designator. If it is absent, the Conversion Aid divides the integer value by 2, giving a character value, and inserts the word CHARACTERS after the integer. The word IS is removed from the MEMORY SIZE clause. It is no longer valid as an optional word. See "SORT Statement" under "Procedure Division" in this appendix for insertion of MEMORY SIZE in the SORT verb.

Unisys COBOL68 defines one module to equal 1000 digits. Unisys COBOL74 defines one module to equal one word, or four digits. To allow a converted program to request the same amount of memory, when memory size is expressed in modules, the Conversion Aid converts modules to words. The integer value is multiplied by 250, and the word MODULES is replaced by the word WORDS.

The phrase TIME 60 is deleted. It denotes that the internal time clocking register is to be maintained as hours, minutes, seconds, and sixtieths of a second, COMPUTATIONAL. The size of the time register in Unisys COBOL68 is 10 digits. In Unisys 1974 COBOL, time is eight digits: HHMMSSSS. This time register is used in converting any subsequent MOVE TIME statements. This conversion is discussed further under "Special Registers" in this appendix.

The word IS is inserted after the word SEGMENT-LIMIT if it is not already present.

SPECIAL-NAMES Paragraph

The statement CURRENCY SIGN IS followed by *L*, /, or = is invalid and is flagged. A valid currency sign replaces the dollar sign ($) as the currency symbol.

Numeric literals and picture strings are handled correctly when DECIMAL POINT IS COMMA appears in this paragraph.

Unisys COBOL68 allows one or more clauses:

[implementor-name IS mnemonic-name]

The implementor name is one of the valid hardware names. Unisys COBOL74 does not have this construct. The clause is deleted and the equated names are used by the Conversion Aid in processing the FILE-CONTROL paragraph.

Unisys COBOL74 requires that special names clauses, if used, be in the following specified order:

ODT IS name

CURRENCY SIGN IS literal

DECIMAL-POINT IS COMMA

Any clause that is out of order is flagged.

INPUT-OUTPUT SECTION

It is possible in Unisys COBOL68 for an INPUT-OUTPUT SECTION header to precede the SOURCE-COMPUTER or OBJECT-COMPUTER paragraphs. If this condition is recognized, the INPUT-OUTPUT SECTION header is deleted from that location. If the FILE-CONTROL paragraph is present and is not immediately preceded by the INPUT-OUTPUT SECTION header, then the INPUT-OUTPUT SECTION header is inserted before it.

An empty FILE-CONTROL paragraph or an empty INPUT-OUTPUT SECTION is flagged as invalid.

FILE-CONTROL Paragraph

The ASSIGN TO hardware name clause can be untouched, changed, or flagged, depending on the hardware name. A mnemonic name that was equated to a hardware name in the SPECIAL-NAMES paragraph is converted to the equated hardware name. That hardware name is then treated as if it were originally present in the ASSIGN clause. The following hardware names are valid in both Unisys COBOL68 and Unisys COBOL74 and are untouched:

- DISK
- DISKPACK
- PRINTER
- PUNCH
- READER
- TAPE

The hardware names shown in Table F–2 are changed as indicated.

Table F-2. Hardware Name Changes

Old Device	New Device
DISC	DISK
TAPE-PE	TAPE 9 CHANNEL PHASE-ENCODED
TAPE-9	TAPE 9 CHANNEL
TAPE-7	TAPE 7 CHANNEL
PT-PUNCH	PUNCH PAPERTAPE
PT-READER	READER PAPERTAPE
SORTER	READERSORTER

All data comm devices or processing units acting as communications devices are flagged as invalid, as shown in Table F–3.

Table F-3. Invalid Communication Devices

ATT-8A1	B-500	B-774	B-874
B-974	B-2500	B-2700	B-2800
B-2900	B-3500	B-3700	B-3800
B-3900	B-4700	B-4800	B-4900
B-9350	B-9352	DCT-2000	DISPLAY-UNIT
FRIDEN-7311	IBM-1030	IBM-1050	O-L-BANKING
TOUCH-TONE	TT-28	TWX	

The devices LISTER and SORTER are flagged as invalid.

A hardware name other than those in the previous list is flagged as not found in the table.

The word SHARED following disk or diskpack indicates a file that can be shared by multiple processors. It is deleted and is converted to the Boolean file attribute SHARED in a VALUE OF clause in the FD.

```
SHARED

    becomes

VALUE OF SHARED IS TRUE
```

The BY clause AREA and CYLINDER options remain valid. The NN option and the FILE option are deleted. The NN option assigns the electronic unit specified by NN. The FILE option distributes files among electronic units according to order of appearance of FD.

The BY NN phrase is converted to the symbolic file attribute SUBSYSTEM. The BY NN phrase appears in a VALUE OF clause in the FD. The NN value is converted to the SUBSYSTEM value as shown in Table F–4.

Table F-4. NN Option Conversions

NN Option	Value of
1 or 01	DSK1
2 or 02	DSK2
3 or 03	DSK3
4 or 04	DSK4
5 or 05	DSK5
6 or 06	DSK6
7 or 07	DSK7
8 or 08	DSK8
Anything else	DEFAULT

The Work and Processor options are both deleted. The Work option specifies that the disk file selected is a work file. The Processor option specifies that multiple processors can share common disk.

Each option is converted to the symbolic file attribute UNIQUENAME. The option appears in a VALUE OF clause in the FD.

WORK

becomes

VALUE OF UNIQUENAME IS WORK

PROCESSOR

becomes

VALUE OF UNIQUENAME IS PROCESSOR

The MULTIPLE UNIT/REEL clause is deleted. This clause is for documentation only. The feature is automatically provided.

The NO TRANSLATION and TRANSLATION NON-STANDARD clauses are deleted. The NO TRANSLATION clause causes bypass of the hardware translator of data between hardware and internal memory. The TRANSLATION NON-STANDARD clause translates lowercase characters transmitted from remote devices to uppercase.

The FORM option is used with PRINTER or PUNCH files. It causes the program to halt and then prints an MCP message. FORMID is deleted from the SELECT statement. It is converted to the Boolean file attribute FORMID, which appears in a VALUE OF clause in the FD. COBOL syntax requires a string value for FORMID; the arbitrary string FORMID is used.

FORM

becomes

VALUE OF FORMID IS "FORMID"

The SAVE option is deleted from the SELECT statement. The SAVE option closes and locks the file if it is still open at end-of-job.

The SAVE option is converted to the symbolic file attribute PROTECTION and appears in a VALUE OF clause in the FD.

SAVE

becomes

VALUE OF PROTECTION IS SAVE

In the RESERVE clause RESERVE integer (or NO) ALTERNATE AREA(S) is changed to RESERVE integer + 1 (or 1) AREA(S). If the file is assigned to SORT DISK, the RESERVE clause is deleted. The maximum number of areas is 9. A number larger than 9 is replaced by 9.

The FILE-LIMIT(S) clause is deleted. This clause specifies logical segments of a file. In SEQUENTIAL access it may modify the sequence of records obtained or placed. In RANDOM access it specifies the limits of records obtained or placed.

The WITH CHECK option is deleted. This option causes the MCP to perform a READ operation after a WRITE operation for diskpack files to check for an error.

The PROCESSING MODE IS SEQUENTIAL clause is deleted. It is for documentation only.

The SYMBOLIC KEY(S) clause is deleted. It is for documentation only.

If the ACTUAL KEY clause is present and the ACCESS MODE clause is not present, then the statement ACCESS MODE IS SEQUENTIAL is inserted before the ACTUAL KEY phrase because in Unisys 1974 COBOL, the ACTUAL KEY phrase is a part of the ACCESS MODE clause.

I-O-CONTROL Paragraph

The MULTIPLE FILE clause is deleted. The multifile ID in this clause identifies the diskpack or tape containing the files named. It is converted to FAMILYNAME in the VALUE OF clause of the FD for each file named in the CONTAINS phrase.

MULTIPLE FILE [DISKPACK/TAPE] "id-literal" CONTAINS file-name-1 file-name-2

becomes

FD file-name-1 VALUE OF FAMILYNAME IS "id-literal"
FD file-name-2 VALUE OF FAMILYNAME IS "id-literal"

The APPLY clause is deleted. It is specified in Unisys COBOL68 for SORTER only.

Unisys COBOL68 allows multiple sentences in the I-O-CONTROL paragraph, i.e., each clause can end with a period. Unisys COBOL74 allows a period only at the end of the paragraph. Periods within the I-O-CONTROL paragraph are deleted.

Data Division

Section names, level indicators (FD and SD), and level numbers (77 and 01) are moved to begin in area A (column 8) if they do not already begin in area A.

Clauses within FDs and SDs are moved to begin in area B (column 12) if they do not already begin in area B.

Level 01 FILLER is changed to FILLER-XX-nn, where nn starts at 01 and is incremented by 1 for each occurrence up to a maximum of 99.

The COPY statement might occur in this context:

FD SOURCE-NAME COPY LIBRARY-FILE

SD SOURCE-NAME COPY LIBRARY-FILE

01 SOURCE-NAME COPY LIBRARY-FILE

In this case, the words FD, SD, or 01 and SOURCE-NAME are deleted from this statement. If the COPY option is set, then the library file is converted and SOURCE-NAME replaces the equivalent name in the first record of the library file.

However, if the source program copies the same library file in more than one place, the library file is converted the first time. Subsequent COPY statements for that file are flagged. The user should correct the syntax and can use the REPLACING option for the FD, SD, or 01 level name.

File Section

The function of the RECORDING MODE IS STANDARD/NON-STANDARD clause is to specify the recording mode for peripheral devices where a choice can be made.

RECORDING MODE is converted to the symbolic file attribute EXTMODE for files assigned to PUNCH, TAPE 7, PUNCH PAPERTAPE, and READER PAPERTAPE. For files assigned to any other device RECORDING MODE is deleted.

RECORDING MODE IS STANDARD is converted as shown in Table F–5.

Table F-5. RECORDING MODE Conversions - STANDARD

Device	Value of EXTMODE
PUNCH	EBCDIC
TAPE 7	BINARY
PUNCH PAPERTAPE	NONSTANDARD
READER	PAPERTAPE NONSTANDARD

RECORDING MODE IS NON-STANDARD is converted as shown in Table F–6.

Table F-6. RECORDING MODE Conversions - NON-STANDARD

Device	Value of EXTMODE
PUNCH	NONSTANDARD
TAPE 7	NONSTANDARD
PUNCH PAPERTAPE	BINARY
READER PAPERTAPE	BINARY

The function of the FILE CONTAINS integer-1 [BY integer-2] RECORDS clause is to indicate the number of logical records in a file.

In an FD and in an SD, the FILE CONTAINS clause is treated as follows: If the file is assigned to a non-mass storage device, FILE CONTAINS is deleted. If the file is assigned to a mass storage device, FILE CONTAINS is converted to both attributes areasize and areas as follows: If the BY phrase is present, then AREAS is the integer preceding the word BY, and AREASIZE is the integer following BY. If the BY phrase is absent, then AREASIZE is the integer and AREAS is set to 1.

FILE CONTAINS integer-1 BY integer-2 RECORDS

becomes

VALUE OF AREASIZE IS integer-2 AREAS IS integer-1

FILE CONTAINS integer-1 RECORDS

becomes

VALUE OF AREASIZE IS integer-1 AREAS IS 1

The BLOCK CONTAINS ... RECORDS/CHARACTERS clause is retained in both an FD and an SD. The function of this clause is to specify the size of a physical record.

In an FD, the LABEL RECORD IS/RECORDS ARE OMITTED clause remains a valid option.

However, in an FD and in an SD, all LABEL RECORD(s) clauses are converted to the symbolic file attribute LABELKIND. The LABEL RECORD(s) clause is converted to VALUE OF LABELKIND as shown in Table F–7.

Table F-7. LABEL RECORD Conversions

Label Record	Value of LABELKIND
STANDARD	MSSTANDARD
OMITTED	OMITTED
USASI	MSUSASII
NON-STANDARD	INSTALLATION

The function of the VALUE OF ID/IDENTIFICATION clause is to define the identification value assigned, or to be assigned, to a file. In an FD and in an SD, this clause is converted to the attribute FILENAME. FILENAME is defined as an external name used to associate a logical file with a physical file.

VALUE OF ID IS

becomes

VALUE OF FILENAME IS

The function of SAVE-FACTOR is to declare the length of time a file is to be saved. In Unisys COBOL68 it can appear only in an FD. It is converted to the attribute SAVEFACTOR. This attribute defines the expiration date as the number of days past the creation date.

SAVE-FACTOR

becomes

VALUE OF SAVEFACTOR

In the DATA RECORD(S) clause, the verb, if present, is made to agree in number with the subject. The converted form is either DATA RECORD IS or DATA RECORDS ARE. This clause is flagged with the warning message, "COMPILER REQUIRES CORRECT NAME(S)".

In the SELECT statement for a sort file, the word SORT is optional following the word ASSIGN in Unisys COBOL68. Unisys COBOL74 requires the word SORT. If SORT is omitted from a SELECT statement and the file name SELECTed appears as the file name in an SD, the SD statement is flagged.

Unisys COBOL68 allows a file to be assigned to SORT and subsequently be declared in an FD. Unisys COBOL74 requires that a file ASSIGNED to SORT be declared in an SD. FD is changed to SD for a file ASSIGNED to SORT.

Unisys COBOL68 allows the VALUE clause in File Section record description entries at levels other than 88 and considers this VALUE to be documentation only. Unisys COBOL74 does not allow this use of VALUE, and the Conversion Aid deletes it.

If the RECORD CONTAINS clause is present in an FD or in an SD, it is flagged with the warning message "COMPILER REQUIRES CORRECT SIZE".

In Unisys COBOL68, the 01 level records of a file assigned to SORTER contain 200 characters. Unisys COBOL74 requires the 01 level records of a file assigned to READERSORTER to contain 390 characters. READERSORTER 01 level records are flagged.

Working-Storage Section

A data item "77 TALLY PIC 9(5) COMP VALUE ZERO." is inserted as the first entry in Working-Storage. The Working-Storage Section Header is inserted before TALLY if this section is not present in the source input.

If the SWITCH option was specified on the & record, then the following switch data-name entries are inserted after TALLY:

77 SW1-XX PIC 9 COMP VALUE ZERO.

through

77 SW8-XX PIC 9 COMP VALUE ZERO.

The MOD clause following the record-name is deleted. MOD in Unisys COBOL68 causes the beginning address of that record to start at the next modulus 1000.

Abbreviations

Abbreviations in the Data Division are changed as shown in Table F–8.

Table F-8. Abbreviation Conversions

Abbreviation	Becomes
PC	PIC
VA	VALUE
SY	SYNC
OC	OCCURS
CMP	COMP
CMP-1	COMP
COMPUTATIONAL-1	COMPUTATIONAL
COMPUTATIONAL-3	COMPUTATIONAL
JS	JUST
BZ	BLANK WHEN ZERO

A change of abbreviation to a longer form can result in line overflow. In this case the line is broken at the beginning of a word or literal and a second line is written.

PICTURE Clause

A PICTURE character string that exceeds 30 characters is flagged.

A PICTURE character string is no longer allowed to contain undefined insertion characters. PICTURE characters defined in Unisys COBOL68 are the following: *$ A B J K P S V X Z 9 0 ( ) , . + − and **. In addition, *C* and *R* are valid in the combination *CR*, *D* is valid in the combination *DB*, and all of the digits are valid repeat characters.

A character newly defined in 1974 ANSI Standard COBOL is stroke or slash (/). Unisys COBOL74 also defines the character *I* as an indicator that the character immediately following it is a simple insertion character. Unless the currency symbol is *I*, the Conversion Aid inserts an *I* in front of each PICTURE string character that is undefined and nondigit. If the currency symbol is *I*, the Conversion Aid replaces each undefined and nondigit PICTURE string character with a stroke. The characters *J* and *K* are no longer valid, but they are handled separately. The following is an example of *I* insertion:

```
999%99#9999

    becomes

999I%99I#9999
```

Unisys COBOL68 allows the number of digit positions represented in the PICTURE string of a numeric data item to range from 1 through 99999. For a numeric-edited data item the range allowed is from 1 through 99. Items used in arithmetic computations cannot exceed 98 digits in length. Unisys COBOL74 is now consistent with Unisys COBOL68 regarding these sizes. A numeric data item larger than 98 digits is flagged that it cannot be used in arithmetic.

A leading *J* or *K* in a PICTURE character string is changed to *S*. If the item is not computational then SIGN TRAILING is added to a J-to-S entry, and SIGN LEADING SEPARATE CHARACTER is added to a K-to-S entry.

A *J* sign within a PICTURE character string is flagged as invalid.

A PICTURE string with + or − between 9s and Ps is flagged. An example of this is 9(7)−PP.

VALUE Clause

If a VALUE string is signed-numeric, it must be associated with a PICTURE character string that begins with *S*. The + or − sign is removed from the first position of a signed numeric literal in a VALUE clause if the associated PICTURE does not begin with S and providing the PICTURE clause precedes the VALUE clause in the data declaration. If PICTURE follows VALUE then the item is flagged.

An entry containing a numeric-edited PICTURE is flagged if the associated VALUE is numeric. The item is determined to be numeric-edited if the PICTURE string contains no *X* or *A* but does contain at least one of the following edit symbols (Z $, . * / 0 + − CR DB) or an undefined character. If the CURRENCY SIGN clause was found to be present and valid, its value replaces the $. The VALUE string is an acceptable nonnumeric literal if it is ZERO, HIGH-VALUE(S), or if the first character is the quotation mark (").

If a PICTURE character string contains an asterisk (*) and is followed by BLANK WHEN ZERO or BZ, then this phrase is deleted.

In Unisys COBOL68, a data-name can be declared with VALUE IS DATE-COMPILED. The date is placed in the data-name in the Julian form YYDDD. In Unisys COBOL74, there is no construct that allows the date to be stored in a data-name at compile time. In a data declaration VALUE DATE-COMPILED is changed to VALUE ZERO.

A numeric data declaration with the scaling character *P* in its PICTURE string is flagged as invalid if its VALUE size is greater than its PICTURE size.

A numeric initial VALUE is enclosed in quotation marks if it is associated with an alphanumeric PICTURE. This includes one or more VALUES for each level 88 condition-name.

If an initial VALUE associated with an alphabetic (A) PICTURE contains any character other than letters or space, it is flagged as invalid.

OCCURS Clause

An item containing OCCURS 0 TIMES is changed to OCCURS 1 TIMES.

In the clause OCCURS integer-1 TO integer-2 TIMES, integer-2 must be greater than integer-1. If it is not greater, then it is replaced with the value of integer-1 + 1. Note that since the minimum value of integer-1 is 1, the minimum value of integer-2 is 2.

In Unisys COBOL74, the DEPENDING clause specifies a variable number of occurrences. If an item contains a DEPENDING clause but not "TO integer-2", then the DEPENDING clause is deleted.

An item containing OCCURS ... TO is flagged if it does not also contain a DEPENDING clause.

Options of the OCCURS clause, namely INDEXED, DEPENDING, ASCENDING, and DESCENDING, cannot be split from OCCURS by other words in the data declaration. If they are split, they are flagged.

In Unisys COBOL68, the ASCENDING/DESCENDING KEY option is for documentation only. Unisys COBOL74 requires that the key(s) named be either the OCCURS or subordinate entries. The ASCENDING/DESCENDING KEY phrase is flagged.

REDEFINES Clause

Unisys COBOL68 allows REDEFINES clauses in successive data declarations at the same level to be chained, that is, name-2 REDEFINES name-1, name-3 REDEFINES name-2, and so on. Unisys COBOL74 requires that the object of the REDEFINES be the original name. The Conversion Aid changes the object of a REDEFINES to the original name if it is not already.

Unisys COBOL68 also allows a qualifier in a REDEFINES clause. Unisys COBOL74 does not allow this, and the Conversion Aid deletes the qualifier.

The word REDEFINES must follow immediately after the data-name. If there are any intervening words, then REDEFINES is flagged.

RENAMES Clause

A RENAMES declaration cannot contain a PICTURE clause. If it does, the Conversion Aid flags it as invalid.

Procedure Division

In the Procedure Division, verbs that are found beginning in area A are moved to area B starting in column 12.

Tests concerning verb options are initiated by the occurrence of the verb and are usually terminated by the occurrence of the item tested for, a period, or another verb, whichever comes first. Exceptions to this termination rule are noted. A period ending a deleted construct is retained if the last retained construct preceding the deleted one does not end with a period. The period is placed up close to the last retained construct if it is on the same or preceding input line, otherwise the period is placed on a separate line.

Certain changes in the Procedure Division, (those concerning ADD and EXAMINE) require that an identifier be recognized. Qualified data-names and subscripted and indexed data-names, as well as 1-word data-names are recognized.

The following verbs and other Procedure Division constructs that are changed, deleted, or flagged are presented in alphabetical order.

ACCEPT and DISPLAY Verbs

The special registers TODAYS-DATE, DATE, and TIME have become options of the ACCEPT verb. They are discussed under "Special Registers" in this section.

In the ACCEPT construct, FROM ODT is no longer valid and is deleted. In the DISPLAY construct, UPON ODT is no longer valid and is deleted.

ADD Verb

Unisys COBOL74 requires that the ADD verb be followed by TO or GIVING. If a period or the reserved words ROUNDED, ON, SIZE, ELSE, OTHERWISE, or another verb is encountered after an ADD with no intervening TO or GIVING, then TO is inserted before the last ADD operand.

The last ADD operand is recognized as the operand ending with a period or the last operand preceding a reserved word. The following are two examples:

```
ADD NAMEA NAMEB.

    becomes

ADD NAMEA TO NAMEB.

ADD NAMEA NAMEB ROUNDED

    becomes

ADD NAMEA TO NAMEB ROUNDED
```

ALTER Verb

In Unisys COBOL68 segmentation does not affect ALTER. Any GO TO paragraph can be ALTERed TO PROCEED TO any paragraph or section in the program. In Unisys COBOL74, modified GO TO statements in independent segments can, under some circumstances, be returned to their initial state or, under other circumstances, remain in their last used state. The Conversion Aid flags all ALTER statements with a warning message.

CLOSE Verb

In V Series Systems usage, the CLOSE option word RELEASE is changed to DISMISS.

Mass Storage Files

For files assigned to mass storage devices, the semantic meaning of the options LOCK and DISMISS has changed from Unisys COBOL68 to Unisys COBOL74. CLOSE LOCK now makes a new mass storage file permanent, but the file cannot be reopened by the executing program. CLOSE CRUNCH includes the features of SAVE and also returns unused space to the system. The Conversion Aid makes changes for mass storage files as shown in Table F–9.

Table F-9. CLOSE Conversions for Mass Storage Files

Old CLOSE	New CLOSE
CLOSE LOCK	CLOSE SAVE
CLOSE RELEASE	CLOSE DISMISS
CLOSE RELEASE CRUNCH	CLOSE CRUNCH
CLOSE LOCK CRUNCH	CLOSE CRUNCH

Non-Mass Storage Files

The following changes apply to files assigned to devices other than mass storage. The option LOCK, in Unisys COBOL68, releases output areas but does not release the device. In Unisys 1974, COBOL the LOCK option releases the device. However, the Conversion Aid does not change CLOSE LOCK.

Unisys COBOL68 allows the options CRUNCH and REMOVE for these files even though they are inconsistent with the device. The Conversion Aid deletes the word CRUNCH and it changes REMOVE to DISMISS. CLOSE statement changes applying to non-mass storage files are shown in Tables F–10 and F–11.

Table F-10. CLOSE Conversions for Non-Mass Storage Files

Old CLOSE	New CLOSE
CLOSE RELEASE	CLOSE DISMISS
CLOSE RELEASE CRUNCH	CLOSE DISMISS
CLOSE LOCK CRUNCH	CLOSE LOCK
CLOSE REMOVE CRUNCH	CLOSE DISMISS
CLOSE REEL CRUNCH	CLOSE REEL
CLOSE REEL REMOVE	CLOSE REEL

CLOSE REEL signifies that the file is a multi-file magnetic tape. The Conversion Aid changes CLOSE REEL syntax as follows: LOCK becomes FOR REMOVAL. The options DISMISS and PURGE are not valid with REEL and are deleted. They are also flagged because their functions are not available. REMOVE is deleted from CLOSE REEL.

Table F-11. CLOSE REEL Conversions for Non-Mass Storage Files

Old CLOSE REEL	New CLOSE REEL
CLOSE REEL LOCK	CLOSE REEL FOR REMOVAL
CLOSE REEL RELEASE	CLOSE REEL
CLOSE REEL PURGE	CLOSE REEL
CLOSE REEL REMOVE	CLOSE REEL

The word UNIT in the CLOSE UNIT statement in Unisys COBOL68 is for documentation only. In Unisys COBOL74, the words REEL and UNIT are synonymous. The Conversion Aid deletes the word UNIT

DIVIDE Verb

The MOD option following the DIVIDE verb is no longer valid and is flagged as invalid.

The REMAINDER option of the DIVIDE verb can no longer be ROUNDED, and ROUNDED is deleted from this option.

ELSE/OTHERWISE Options

OTHERWISE is no longer a reserved word and is no longer interchangeable with ELSE. It is replaced by ELSE except where the ELSE/OTHERWISE option is flagged as invalid.

The ELSE/OTHERWISE option of the verbs READ, WRITE, ADD, COMPUTE, DIVIDE, MULTIPLY, and SUBTRACT is no longer valid. If ELSE or OTHERWISE is encountered following one of these verbs, it is flagged as invalid. A period terminates the test for ELSE/OTHERWISE.

Unisys COBOL68 allows null THEN syntax and null ELSE or OTHERWISE syntax in IF, arithmetic, and I/O statements. Unisys COBOL74 does not allow this. An ELSE or OTHERWISE is null if it is followed by a period. It is deleted. A null THEN is flagged with the following message:

```
IMPERATIVE STATEMENT MISSING
```

ENTER Construct

Unisys COBOL68 provides for the use of an alternate language with the construct ENTER SYMBOLIC ... ENTER COBOL. Unisys COBOL74 does not support this construct. The Conversion Aid flags ENTER SYMBOLIC as invalid.

EXAMINE Format

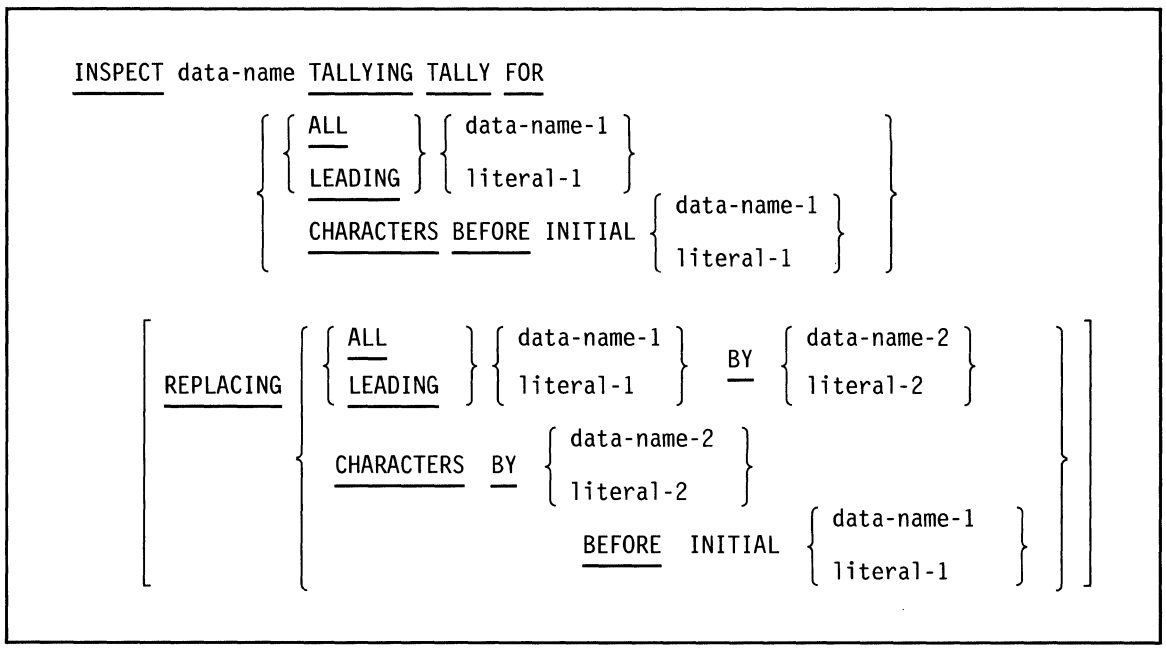
In COBOL74 the INSPECT format replaces the EXAMINE format. The data-name TALLY is initialized by a MOVE ZERO TO TALLY statement whenever the TALLYING option is used or when the REPLACING option is used with a qualified data-name. When initialized, a previously existing value of TALLY is no longer available. New statements are begun on new lines. The following are the old and new formats.

Formats with the TALLYING option:

Old Format

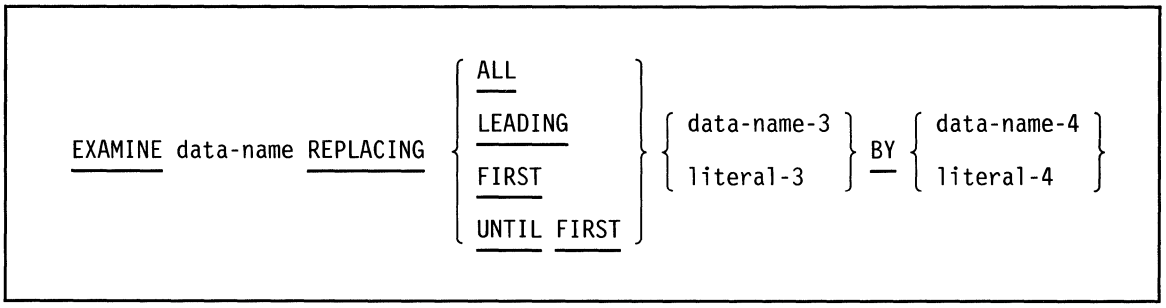
```
EXAMINE  data-name  TALLYING    { ALL       }   { data-name-1 }
                               { LEADING   }   { literal-1   }
                               { UNTIL FIRST }

      [ REPLACING  BY   { data-name-2 } ]
                        { literal-2   }
```

New Format

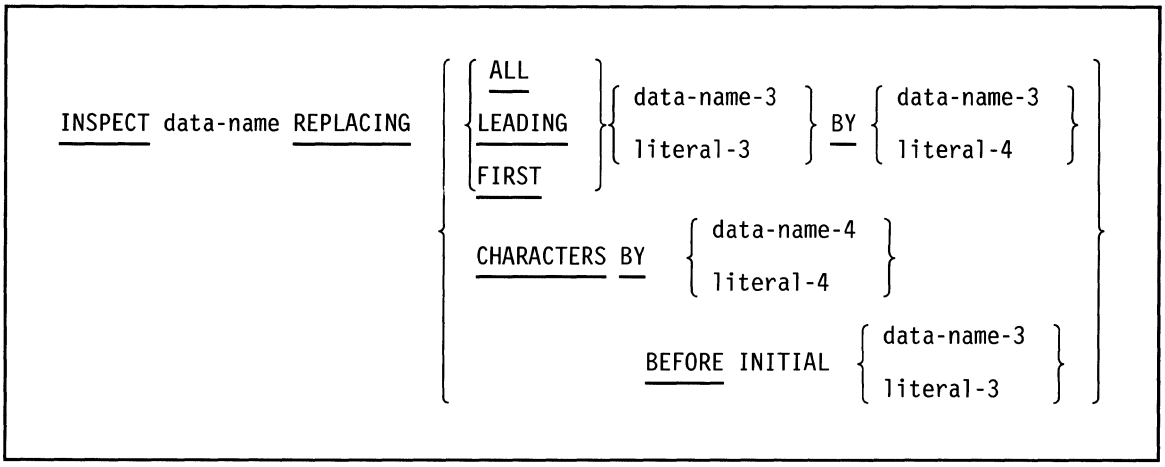


Formats with the REPLACING option:

Old Format



New Format



EQUALS Verb

The word EQUALS is changed to EQUAL.

FILL Statement

The FILL in Unisys COBOL68 passes data from one program to another when both programs are operating in the same multiprogramming mix.

The FILL statement is converted to SEND, RECEIVE, or ACCEPT, as follows:

- Data sending becomes SEND.
  - FILL data-name INTO program-name [PROCEED TO paragraph]

    becomes

    SEND program-name FROM data-name [ON EXCEPTION GO TO paragraph]
  - FILL data-name INTO {TOP/BOTTOM} [PROCEED TO paragraph]

    becomes

    SEND TO {TOP/BOTTOM} data-name [ON EXCEPTION GO TO paragraph]
- Data receiving becomes RECEIVE.
  - FILL data-name FROM program-name [PROCEED TO paragraph]

    becomes

    RECEIVE program-name INTO data-name [ON EXCEPTION GO TO paragraph]

- FILL data-name FROM {TOP/BOTTOM} [PROCEED TO paragraph]

  becomes

  RECEIVE FROM {TOP/BOTTOM} data-name [ON EXCEPTION GO TO paragraph]

- Data inquiry becomes ACCEPT.
  - FILL data-name FROM POLL

    becomes

    ACCEPT data-name MESSAGE COUNT

A data comm FILL statement is flagged as invalid and is not changed.

IF Statement

A compound conditional introduced by IF can have an implied subject or an implied subject and operator. It must, however, have a connective (AND or OR) between object operands. An IF statement containing consecutive operands is either flagged or changed. If the operands are simple data-names and only one connective is missing, then the connective is inserted. If the operands are subscripted, indexed, or qualified, or they form a list, then the statement is flagged.

In an IF statement relation phrase, the word THEN following the word LESS or the word GREATER is changed to THAN.

INTERROGATE Statement

The function of the INTERROGATE FILE statement is to test for the presence of a file on disk or diskpack. It is converted to CALL SYSTEM INTERROGATE as follows:

INTERROGATE FILE name-1 INTO name-2

becomes

CALL SYSTEM INTERROGATE USING name-1 INTO name-2

If the DISKPACK option is used, the statement is also flagged. In Unisys COBOL68, name-1 includes both file-name and family-name. Unisys COBOL74 syntax requires that DISKPACK name-1 be written as file-name DISKPACK family-name.

Literals

The verbs MOVE, IF, PERFORM, and SEARCH can use the reserved word ALL followed by a literal. The verb EXAMINE can use the reserved words ALL, LEADING, [UNTIL] FIRST, and BY followed by a literal. This literal is restricted to the types nonnumeric and hexadecimal. Numeric literals following these reserved words are changed to nonnumeric by being enclosed in quotation marks. Hexadecimal literals following these reserved words are flagged because they are treated as alphanumeric data rather than as numeric.

A numeric literal in Unisys COBOL68 can range up to 98 digits in size if it is used in arithmetic statements, or 160 digits otherwise. Unisys COBOL74 is now consistent with Unisys COBOL68 regarding size of literals. A numeric literal greater than 98 digits is flagged that it cannot be used in arithmetic.

OPEN Statement

The option word INPUT-OUTPUT is changed to I-O.

The OPEN O-I file-name option is flagged as invalid.

Unisys COBOL68 allows OPEN options REVERSED and NO REWIND for a disk file, and issues WARNING messages if they are present. Unisys COBOL74 issues syntax errors for these constructs. The words REVERSED and NO REWIND are deleted from an OPEN statement for a disk file.

READ Statement

In Unisys COBOL74, the AT END option applies to SEQUENTIAL ACCESS files, and the INVALID KEY option applies to RANDOM ACCESS files. ACCESS MODE is SEQUENTIAL unless it is declared as RANDOM in the SELECT statement for the file. READ AT END is changed to READ INVALID KEY for RANDOM ACCESS files. READ INVALID KEY is changed to READ AT END for SEQUENTIAL ACCESS files.

In the READ construct, if the file is assigned to mass storage and ACCESS MODE IS RANDOM, then either INVALID KEY or a USE procedure is required. If INVALID KEY is not present, the statement is flagged with a warning message.

READ file-name INTO record-name is valid if record-name is declared in Working-Storage or in an FD other than the one being READ. If record-name is declared in the file-name being read, then the INTO phrase is deleted.

In a READ statement, AT END or INVALID KEY must be followed by one or more imperative statements. An IF statement or another READ statement that follows AT END or INVALID KEY is flagged. A period terminates the original READ statement.

The phrases (WITH) LOCK, LOCK-ONLY, and LOCK-WAIT in Unisys COBOL68 apply to shared disk and diskpack MCP usage. READ statements containing these words are converted to system dependent intrinsics (SDIs). WITH LOCK reads and locks a block of data. LOCK-ONLY locks a block of data but does not transfer it. LOCK-WAIT reads and locks a block of data after waiting for it to become available if it is currently locked. READ statements are converted as follows:

```
READ file-name (RECORD) (WITH) LOCK

    becomes

CALL "INTRINSIC READLOCK" USING file-name


READ file-name (RECORD) (WITH) LOCK-ONLY

    becomes

CALL "INTRINSIC READLOCKONLY" USING file-name


READ file-name (RECORD) (WITH) LOCK-WAIT

    becomes

CALL "INTRINSIC READLOCKWAIT" USING file-name
```

The phrase INVALID KEY is converted to ON OVERFLOW. If INVALID KEY is not present, a flag message is issued that ON OVERFLOW or a USE procedure is required.

SEARCH Statement

Unisys COBOL68 allows SEARCH data-name-1 VARYING to vary two index-names or a data-name and an index-name. Unisys COBOL74 allows only one name to be varied. SEARCH data-name-1 VARYING two names is flagged as invalid.

The WHEN phrase of SEARCH ALL specifies either an equal condition or a condition-name. The subject of the condition or the data-name associated with the condition-name must be referenced in the KEY IS phrase of the OCCURS clause of the identifier being SEARCHed. The object of the condition cannot be so referenced. All SEARCH ALL statements are flagged to alert the user to the existence of these new rules.

Section Headers

In Unisys COBOL68 sectioning can begin with any paragraph in the Procedure Division, not necessarily at the beginning of the Division. Unisys COBOL74 requires that if one paragraph is in a section, then all paragraphs must be in sections.

If the Procedure Division does not begin with DECLARATIVES or with a Section Header, then COBOL74-HEADER SECTION. is inserted as the first record after the Division Header.

If DECLARATIVES are present, they consist of one or more sections. The record following END DECLARATIVES must be a new Section Header. If none is present, then NON-DECLARATIVES SECTION is inserted after END DECLARATIVES.

SEEK Statement

The WITH KEY CONVERSION option of the SEEK verb is deleted. This option in Unisys COBOL68 indicates that USE FOR KEY CONVERSION in the DECLARATIVES section is to be executed prior to the SEEK. USE FOR KEY CONVERSION is invalid in Unisys COBOL74 and is flagged.

The phrases (WITH) LOCK, LOCK-ONLY, and LOCK-WAIT have the same functions as in READ, and they are converted similarly. See "READ Statement" in this appendix. Because of program limitations, if the CONVERSION phrase is present, the statement can be flagged rather than converted.

SORT Statement

The WITH "literal" option of the SORT verb is flagged as invalid and requiring manual change. This option invokes the key translation option of the sort intrinsic. The SORT verb has a new COLLATING SEQUENCE option.

The clause MEMORY SIZE IS integer CHARACTERS from the OBJECT-COMPUTER paragraph is inserted into the SORT statement. If integer is not in CHARACTER units in the OBJECT-COMPUTER paragraph, it is converted for use in the SORT statement.

The word RELEASE is changed to DISMISS in the GIVING phrase. The word WITH is no longer valid in the USING and GIVING phrases and is deleted.

Special Registers

The special registers in Unisys COBOL68 are: TODAYS-DATE, DATE, TIME, and CORE-SIZE. In Unisys COBOL74, TODAYS-DATE, DATE, TIME and TIMER are options of the ACCEPT verb.

TODAYS-DATE Register

The register TODAYS-DATE contains the current calendar date in the 6-digit format month, day, year (MMDDYY). The Conversion Aid converts TODAYS-DATE as follows:

```
MOVE TODAYS-DATE TO identifier-1.

    becomes

ACCEPT identifier-1 FROM TODAYS-DATE.
```

If TODAYS-DATE occurs in any context other than as the sending field in a MOVE statement, it is flagged as invalid.

DATE Register

The current register DATE contains the current Julian date in the 5-digit format year, day (YYDDD). The equivalent register in Unisys COBOL74 is DAY. The following change is made:

```
MOVE DATE TO identifier-1.

    becomes

ACCEPT identifier-1 FROM DAY.
```

If DATE occurs in any context other than as the sending field in a MOVE statement, it is flagged as invalid.

TIME Register

In Unisys COBOL68, TIME is maintained in milliseconds as a 10-digit COMPUTATIONAL field. If TIME 60 is declared in the OBJECT-COMPUTER paragraph, then TIME is maintained as hours, minutes, seconds, and sixtieths of a second. Unisys COBOL74 has two time registers. TIME is an 8-digit field composed of the data elements hours, minutes, seconds, and hundredths of a second (HHMMSSSS). TIMER is composed of the object computer's interval timer, in COBOL, an unsigned elementary numeric integer data item ten digits in length. If TIME 60 was declared in the OBJECT-COMPUTER paragraph, the following change is made:

```
MOVE TIME TO identifier-1.

    becomes

ACCEPT identifier-1 FROM TIME.
```

and the statement is flagged with the following message:

TIME FIELD IS 8 DIGITS HHMMSSSS.

If TIME 60 was not declared, then

MOVE TIME TO identifier-1.

becomes

ACCEPT identifier-1 FROM TIMER.

All other occurrences of TIME are flagged as invalid.

CORE-SIZE Register

The current register CORE-SIZE returns the program's core allocation during execution. There is no equivalent register in Unisys COBOL74. A MOVE CORE-SIZE ... statement is deleted. If CORE-SIZE occurs in any context other than as the sending field in a MOVE statement, it is flagged as invalid.

STOP Statement

The sign of a numeric literal immediately following the word STOP is removed.

Switches (Internal Program Switches)

In Unisys COBOL68, there are eight program switches provided automatically (SW1 through SW8). They can be used in any way a data-name can be used, for instance, as operands of a MOVE or arithmetic statement.

Unisys COBOL74 provides for a restricted use of these eight switches, namely, testing a switch-status condition. The switch-name and the "on" or "off" value associated with the condition must be named in the SPECIAL-NAMES paragraph of the Environment Division. The switch then can be tested by referencing its condition name.

If SWITCH was not specified in the & record, then all references to SW1...SW8 are considered to be references to program switches, and they are flagged as requiring manual change. If SWITCH was specified in the & record, then all references to SW1...SW8 are considered to be references to data-names. Data-names SW1-XX through SW8-XX are inserted in Working-Storage as described earlier under Working-Storage, and in the Procedure Division, the names SW1...SW8 are converted to SW1-XX... SW8-XX.

TRACE Statement

The function of TRACE in Unisys COBOL68 is to create documentation of all normal and/or control mode processing and send it to a line printer. In Unisys COBOL74, TRACE is handled as a system utility through inter-program

communication extensions of the CALL verb. TRACE statements are converted as shown in Table F–12.

Table F-12. TRACE Statement Conversions

TRACE Statement	CALL Statement
TRACE 0	CALL SYSTEM TRACE-OFF
TRACE 1, 2, OR 3	CALL SYSTEM TRACE-ON
TRACE 20 OR 22	CALL SYSTEM DUMP

UNLOCK Statement

The function of UNLOCK in Unisys COBOL68 is to make a shared disk or diskpack file available to other programs. It is converted to a system dependent intrinsic (SDI) as follows:

```
UNLOCK  record-name

    becomes

CALL "INTRINSIC  UNLOCK"  USING  record-name
```

USE Statement

In the USE AFTER STANDARD ERROR PROCEDURE option, the following changes occur:

- The reserved word AFTER is required. If not already present, it is inserted.
- The word INPUT-OUTPUT is changed to I-O.
- The option O-I is flagged as invalid.

The USE AT END OF PAGE ... option is no longer valid and is flagged as invalid. This option allows the object program to branch automatically to a central user routine at the time a 12-punch is read in the carriage control tape.

The USE FOR KEY CONVERSION ... option is no longer valid and is flagged as invalid. This option is used in conjunction with a SEEK WITH KEY CONVERSION statement.

The USE ... LABEL PROCEDURE option is flagged as invalid. This option provides for checking data on input magnetic tape labels and inserting data on output magnetic tape labels.

The USE ON STALEMATE option is no longer valid and is flagged as invalid. This option specifies the action to be taken when two processors sharing disk are accessing the same file, and each is trying to read a record locked by the other processor.

WRITE Statement

WRITE BREAKOUT is flagged as an error.

The AFTER (BEFORE) ADVANCING TO CHANNEL integer option is either changed or flagged. If integer equals 1, then CHANNEL 1 is changed to PAGE; otherwise, CHANNEL integer is flagged. CHANNEL followed by an integer other than 1 must be manually changed to a mnemonic-name that must be defined in the SPECIAL-NAMES Paragraph.

The TO ERROR (AUXILIARY) phrase is deleted. This option causes the record being written to go to the ERROR or AUXILIARY stacker.

In Unisys COBOL68, the default action of a WRITE statement to a printer file where the ADVANCING phrase is not used is BEFORE ADVANCING 1 LINE. In Unisys COBOL74, the default is AFTER ADVANCING 1 LINE. The Conversion Aid inserts the phrase BEFORE ADVANCING 1 LINE into a WRITE statement of a record to a file assigned to PRINTER if an ADVANCING option is not already present.

In the WRITE construct, if the file is assigned to mass storage and ACCESS MODE IS RANDOM, then either INVALID KEY or a USE procedure is required. If INVALID KEY is not present, the statement is flagged with a warning message.

The phrase (WITH) LOCK in Unisys COBOL68 applies to shared disk and diskpack MCP usage. WRITE WITH LOCK is specified for a record locked by the same program. It writes the record but does not unlock it. WRITE WITH LOCK is converted to a system dependent intrinsic (SDI) as follows:

```
WRITE record-name (WITH) LOCK

    becomes

CALL "INTRINSIC WRITELOCK" USING record-name.
```

ZIP Statement

The ZIP statement in Unisys COBOL68 causes the MCP to execute a control instruction, such as scheduling execution of another program. ZIP data-name is converted to CALL SYSTEM ZIP USING data-name.

All Divisions

The following changes pertain to more than one division.

BCL Plus Sign

An invalid BCL plus sign (+) having a hexadecimal code of 12–0 occurs in some COBOL programs. It appears as a plus sign on a BCL printer and as a left brace ({) on an EBCDIC printer and on a CRT terminal. Each occurrence of this BCL plus sign is converted to a valid EBCDIC plus sign having a hexadecimal code of 4E.

NOTE Statement

NOTE statements in all divisions are recognized from the beginning of the word NOTE through the first encountered period. They are flagged as comments by inserting * in column 7. A NOTE statement on the same line as compilable source coding is moved to a new line. NOTE paragraphs in the Procedure Division are recognized by the word NOTE as a paragraph name or as the first word following a paragraph name. NOTE is recognized as a paragraph-name if it begins in Margin A and the preceding line ends with a period. Neither the NOTE paragraph-name nor the NOTE paragraph is required to end with a period. Every line of a NOTE paragraph is flagged as comments by inserting an asterisk (*) in column 7.

New Reserved Words

The Conversion Aid program recognizes new reserved words everywhere in the program except in comment entries. In the Identification Division all entries following paragraph names are comment entries except for program-name.

In the Environment Division, SOURCE-COMPUTER name is a comment entry.

New reserved words are appended with -XX. They are listed in three groups: ANSI reserved, ANSI Report Writer reserved, and Unisys reserved. These are shown in Table F-13, Table F-14, and Table F-15.

Table F-13. ANSI Reserved Words

ALSO	EGI	QUEUE
CALL	EMI	RECEIVE
CD	END-OF-PAGE	REFERENCES
CHARACTER	EOP	RELATIVE
CLOCK-UNITS	ESI	REMOVAL
CODE-SET	EXCEPTION	REWRITE
COLLATING	EXTEND	SEGMENT
COMMUNICATION	INITIAL	SEND
COUNT	INSPECT	SEPARATE

continued

Table F-13. ANSI Reserved Words (cont.)

DAY	LENGTH	SEQUENCE
DEBUG-CONTENTS	LINAGE	SORT-MERGE
DEBUG-ITEM	LINAGE-COUNTER	STANDARD-1
DEBUG-LINE	LINE	START
DEBUG-NAME	LINKAGE	STRING
DEBUG-SUB-1	MERGE	SUB-QUEUE-1
DEBUG-SUB-2	MESSAGE	SUB-QUEUE-2
DEBUG-SUB-3	NATIVE	SUB-QUEUE-3
DEBUGGING	OFF	SUPPRESS
DELETE	ORGANIZATION	TABLE
DELIMITED	OVERFLOW	TERMINAL
DELIMITER	POINTER	TEXT
DESTINATION	PRINTING	TRAILING
DUPLICATES	PROCEDURES	UNSTRING
DYNAMIC	PROGRAM	

Table F-14. ANSI Report Writer Reserved Words

CF	INDICATE	REPORT
CH	INITIATE	REPORTING
CODE	LAST	REPORTS
COLUMN	LIMIT	RESET
CONTROLS	LIMITS	RF
DE	LINE-COUNTER	RH
DETAIL	NUMBER	SOURCE
FINAL	PAGE-COUNTER	STATUS
FOOTING	PF	SUM
GENERATE	PH	TERMINATE
GROUP	PLUS	TYPE
HEADING	RD	

Table F-15. Unisys Reserved Words

ASCII	HIGH	TB
ATTRIBUTE	INPLACE	TIMER
AUDIT	INQUIRY	TODAYS-NAME
AVAILABLE	INSERT	TRACE-OFF
BEGIN-TRANSACTION	INVOKE	TRACE-ON
CARDS	MODIFY	TRANSACTION
CASSETTE	MODULE	TRANSCEIVE
CHANGE	NO-AUDIT	TRUE
COLUMNS	NONE	UPDATE
COMMON	NULL	V-300
CRCR-INPUT	ODT	V-400
CRCR-OUTPUT	ODT-INPUT-PRESENT	V-500
CREATE	OFFER	VIA
CURRENT	OFFSET	WHERE
DATA BASE	PAPERTAPE	WRITE-OK
DB	PHASE-ENCODED	
DEFAULT	PORT	
DISMISS	PRIOR	
DMERRORS	READ-OK	
DMSTATUS	RECREATE	
DUMP	REMOTE	
EBCDIC	STACK	
END-TRANSACTION	STOQ-INPUT	
FALSE	STOQ-OUTPUT	
FIELD	SYSTEM	
FIND	TAG-KEY	
FORMS	TAG-SEARCH	
FREE	TAPES	
GCR		

Separators

A space is inserted as a separator as follows:

- In the Procedure Division, a space is inserted as follows:
  - After a period ending a sentence if a space is not already there.
  - After a period ending a paragraph name if a space is not already there.
  - Before and/or after a relational operator character (= < >) if it is not already separated by at least one space from its operands.
  - Before and/or after an arithmetic binary operator (+ * / **) and before a unary + preceding a data-name if a separator is not already present. The minus sign (-) already requires spacing to distinguish it from the hyphen.

  Unisys COBOL68 supports relative indexing, but does not require a space between the sign and the literal in a relative index. Unisys COBOL74 does require a space. The Conversion Aid is unable to distinguish between a relative index and a signed literal index. It emits a flag message when it detects an identifier with parentheses containing string with a leading sign character.
- In the Environment, Data, and Procedure Divisions, a space is inserted as follows:
  - Before a beginning quote if it is not immediately preceded by a space or left parenthesis
  - After an ending quote if it is not immediately followed by a space, comma, semicolon, period, or right parenthesis
  - After a comma or a semicolon used as a separator if a space is not already present

THEN Separator

Unisys COBOL68 allows the word THEN as a separator in constructs other than IF statements. Unisys COBOL74 allows THEN only in IF statements. The Conversion Aid deletes the word THEN from any construct other than an IF statement.

Figurative Constants

IN Unisys COBOL68, HIGH-VALUE represents 99 and LOW-VALUE represents blanks. In Unisys COBOL74, HIGH-VALUE represents FF and LOW-VALUE represents 00. All occurrences of HIGH-VALUE and LOW-VALUE are flagged; they can no longer be assigned to a numeric data item.

Blank Lines

Lines that are blank in columns 7 through 72 can be deleted if the total number of lines to be written at one time exceeds nine. This total includes the last

preceding nonblank line and any associated new lines and message lines. Doing so does not affect comment lines with an asterisk (*) or slash (/) in column 7.

COPY Statement

Unisys COBOL68 allows the file-name in a COPY statement to be in quotation marks and to contain lowercase letters. Unisys COBOL74 does not allow quotation marks and restricts characters to uppercase letters, digits, and hyphens. The Conversion Aid removes quotation marks and flags invalid characters.

Unisys COBOL68 provides for the creation of library files during compile time by use of source records with an L in column 7 and the library-name in area A. Unisys COBOL74 does not have this facility. An L in column 7 is converted to an *, thus making the record a comment.

If the COPY option is set and the source program copies the same library file in more than one place, the library file is converted only the first time.

Handling of COPY syntax in the Data Division is discussed under "Data Division" in this section.

Periods

The Conversion Aid removes redundant periods.

Reference Format

All division headers are moved to begin in Area A (column 8) if they do not already begin in there.

A nonnumeric literal that starts in area A is either moved to area B or flagged. If it ends on the same line and is less than 60 characters in length, it is moved to begin in area B (column 12); otherwise, it is flagged.

Data comm Constructs

In Unisys COBOL68, data comm constructs in the Procedure Division activate the data communications equipment, as defined by the ASSIGN to hardware-name clause. Data comm constructs have not been specified for Unisys COBOL74. All data comm constructs are flagged as invalid. They are listed in the following list along with the rule by which they are recognized as being data comm constructs.

ACCEPT

The following is the syntax:

```
ACCEPT data-name-1 FROM { literal-1   }
                        { data-name-2 }
```

The word following FROM cannot be ODT or a mnemonic-name equated to ODT in the SPECIAL-NAMES paragraph.

CLOSE

The following is the syntax:

```
CLOSE datacomm-file-name
```

The file must be assigned to a data comm device. The verb CLOSE can be followed by one or more file-names, any of which can be data comm files.

DISABLE

The following is the syntax:

```
DISABLE
```

This is a unique verb.

DISPLAY

The following is the syntax:

```
DISPLAY { literal-1   } [ { literal-1   } ... ] UPON { literal-2   }
        { data-name-1 }  [ { data-name-2 }       ]    { data-name-3 }
```

The word following UPON cannot be ODT or a mnemonic-name equated to ODT in the SPECIAL-NAMES paragraph.

ENABLE

The following is the syntax:

```
ENABLE file-name
```

The file must be assigned to a data comm device.

FILL

The following is the syntax:

```
FILL file-name
```

The file must be assigned to a data comm device.

INTERROGATE

The following is the syntax:

```
INTERROGATE [END-TEXT] file-name
```

The word following INTERROGATE cannot be FILE.

OPEN

The following is the syntax:

```
OPEN option datacomm-file-name
```

The file must be assigned to a data comm device.

READ

The following is the syntax:

```
READ file-name
```

The file must be assigned to a data comm device.

WAIT

The following is the syntax:

```
WAIT [ UNTIL { literal  } ]
     [       { dataname } ]
```

The verb WAIT exists as the only word in the construct; therefore, it is followed by a period, by another verb, or by the word UNTIL.

WRITE

The following is the syntax:

```
WRITE record-name
```

Record-name is a level 01 record-name in a file assigned to a data comm device.

WRITE-READ

The following is the syntax:

```
WRITE-READ
```

This is a unique verb.

WRITE-READ-TRANS

The following is the syntax:

```
WRITE-READ-TRANS
```

This is a unique verb.

WRITE-TRANS-READ

The following is the syntax:

```
WRITE-TRANS-READ
```

This is a unique verb.

Reader-Sorter and Lister Constructs

In Unisys COBOL68, reader-sorter and lister constructs in the Procedure Division activate READERSORTER and LISTER equipment, as defined by the ASSIGN to hardware-name clause. Lister constructs have not been specified for Unisys COBOL74. The Conversion Aid flags them as invalid. Reader-sorter constructs have been implemented in Unisys COBOL74. The Conversion Aid flags these constructs as requiring manual conversion. The following list presents the constructs along with the rule by which they are recognized as reader-sorter or lister constructs.

CONTROL - Reader-sorter

The following is the syntax.

```
CONTROL (4 or 6)
```

Verb is unique. 4 or 6 specifies reader-sorter.

OPEN - Reader-sorter

The following is the syntax.

```
OPEN INPUT [ { DEMAND } ] file-name-1 [ file-name-2 ]
           [ { FLOW   } ]
```

The file must be assigned to the reader-sorter.

READ - Reader-sorter

The following is the syntax.

```
READ [FLOW] file-name
```

The file must be assigned to the reader-sorter.

SELECT - Reader-sorter

The following is the syntax.

```
SELECT
```

Verb is unique.

USE - Reader-sorter

The following is the syntax.

```
USE { ON  } SORTER
    { FOR }
```

USE is followed by SORTER, ON SORTER, or FOR SORTER.

CONTROL - Lister

The following is the syntax.

```
CONTROL (1,2, or 3)
```

Verb is unique. 1,2, or 3 specifies lister.

ENABLE - Lister

The following is the syntax.

```
ENABLE file-name
```

The file must be assigned to lister.

WRITE - Lister

The following is the syntax.

```
WRITE record-name NOT-READY
```

Record-name must be a level 01 record-name in a file assigned to lister.

Exception Conditions

The following paragraphs explain exception conditions that can occur during the use of the COFLTR program.

& Record Errors

Error messages pertaining to the & record can be divided into two groups: those relating to an individual field on the & record and those relating to the & record as a whole.

Error messages relating to a field are the following:

INVALID OPTION WORD

INVALID RESEQUENCE START

+ NOT FOLLOWED BY DIGIT

INVALID RESEQUENCE INCREMENT

Error messages relating to an & record as a whole are the following:

NO INPUT OPTION

MULTIPLE INPUT OPTIONS

NO OUTPUT OPTION

MULTIPLE OUTPUT OPTIONS

NO LIST OPTION

MULTIPLE LIST OPTIONS

MULTIPLE OBJECT-COMPUTER OPTIONS

The following messages appear for the reasons specified:

& CARD OPTION ERROR

If one or more of the above error messages apply to the last & record.

CARD COLUMN 7 NOT &, $, *, /, OR BLANK

If an unidentified record precedes the beginning of the COBOL program.

EOF FOUND FOR CARDFILE, INPUT OPTION IS CARDIN

If the input file ends where the beginning of the COBOL program is expected.

PROGRAM EXECUTION STOPPED

The final Input Module error message.

LIST OPTION SYNTAX OVERRIDES OUTPUT OPTION

A warning message that does not stop program execution.

Input File Validation

An important requirement in using the Conversion Aid is that the COBOL program to be converted must be free from syntax errors.

The Conversion Aid recognizes division headers in the input file and checks for their occurrence in proper sequence. The purpose of this check is to recognize a non-COBOL file or a damaged file, which can inadvertently be read into the Conversion Aid. If any division headers are missing or are out of sequence, then the following message is written at the end of the listing just before the summary:

*** THE PRECEDING LISTING IS NOT RECOGNIZED AS A COBOL PROGRAM

Expansion of Source Coding

An input line of source code can be expanded to a maximum of six output lines consisting of source coding and messages. If the code exceeds six lines, the following line is written on the listing preceding the overflow line:

****** NEW SOURCE CODE BUILD AREA EXCEEDED ******

File Names

File names in SELECT statements are recognized and stored in a table whose maximum size is 40 entries. Names in this table are used in converting FD entries, SD entries, CLOSE, READ, and WRITE statements, and in recognizing certain data comm and reader-sorter and lister constructs. If a source program selects more than 40 files, then each file over 40 causes the following message to appear on the listing:

****** FILE NAMES EXCEED LIMIT OF 40 - - - FILE file-name OMITTED FROM TABLE ******

Subsequent references to an omitted file-name are flagged.

Sequence Numbers

Unless the NORESEQ option is specified on the & record, all programs are automatically resequenced using default values starting at 100 and incrementing

by +10 or by values specified on the & record. If a calculated sequence number causes overflow of the six-character field, then the following message appears on the listing:

****** SEQUENCE NUMBER OVERFLOW - SUBSEQUENT LINES ARE WRITTEN WITH SEQUENCE NUMBER 999999 ******

If NORESEQ is specified on the & record, then automatic resequencing is suppressed. With this feature, an inserted line is given a sequence number of the last output sequence number incremented by 1. In some cases doing so can cause duplicate or disordered sequence numbers. If this occurs, the following message appears on the listing:

****** INSERTED LINE(S) CAUSE SEQUENCE NUMBER DISORDER ******

Recognizing Ending Quote

If the Conversion Aid recognizes a beginning quote and does not recognize an ending quote on the same line, it sets a switch indicating that the next line should continue the quote. If the next line has no continuation indicator in column 7 or its text does not begin with a quote, then the Conversion Aid turns off program switch and the following message appears on the listing:

****** ENDING QUOTE NOT FOUND, ASSUMED ******

Record Names

In the File Section of the Data Division, the Conversion Aid stores all record-names in a table whose maximum size is 80 entries. The Conversion Aid uses the names in this table in converting READ and WRITE statements and in recognizing LISTER and data comm WRITE statements. If more than 80 record-names are declared in the File Section of a program, then each record-name over 80 causes the following message to appear on the listing:

****** TABLE RECORD-NAMES EXCEED LIMIT OF 80 - record-name OMITTED FROM TABLE ******

The Conversion Aid flags subsequent references to an omitted record-name.

Equated Hardware Names

In the SPECIAL-NAMES paragraph in Unisys COBOL68, hardware-names can be equated to mnemonic-names. The Conversion Aid deletes these clauses and stores the name pairs in a table whose maximum size is five pairs. If the source program contains more than five hardware-name equate clauses, then the following message appears in the listing:

****** EQUATED HARDWARE NAME ENTRIES EXCEED LIMIT OF FIVE ******

Output From The COBOL Conversion Aid

The following describes the output from COFLTR.

Blocking of Files

Source file input can be blocked 1 through 10 or 20 if it is from disk, diskpack, or tape. It is blocked 1 if it is from cards. Source file output is blocked 9 if it is to disk, diskpack, or tape. It is blocked 1 if it is to punch. Library file input is from disk or diskpack. It is blocked 1 through 9. Library file output is to disk or diskpack. It is blocked 9.

Output Listing

The following is the title of the first page of the listing:

THIS SOURCE LISTING IS OUTPUT FROM THE COBOL Conversion Aid.

On the output listing, COFLTR writes messages to the right of the text of all lines that it has added, changed, or deleted. A message appears for a deleted line, but not a sequence number or text.

Certain messages can be prefixed by the word CHANGE or DELETE. If a whole line is deleted, the word DELETE is used. If an item is deleted but other source coding remains on the line, the word CHANGE is used. In this way, the number of CHANGE and DELETE messages is keyed to the SOURCE CODE SUMMARY record counts at the end of the output listing. If a deleted entry consists of more than one line, the listing includes a second message stating the total number of lines deleted for that entry.

All FLAG messages end with the symbols #### to enhance visibility. FLAG messages indicate one of the following conditions:

- A construct is invalid but cannot be corrected by the Conversion Aid and, therefore, requires manual change.
- A construct can produce results that differ in execution of Unisys COBOL74 from the results in execution of Unisys COBOL68.
- New rules of usage apply to the construct.

List of Messages

The following Conversion Aid output messages are grouped by division.

The following messages can appear in more than one Division:

```
CHANGE - DIVISION NAME MOVED TO AREA A
CHANGE - SECTION NAME MOVED TO AREA A
CHANGE - PARAGRAPH NAME MOVED TO AREA A
CHANGE - * PLACED IN COLUMN 7
DELETE - BLANK LINE(S)       NUMBER = ZZZZZZ
CHANGE - NEW RESERVED WORD(S), -XX APPENDED
FLAG   - HIGH-VALUE REPRESENTATION IS FF ####
FLAG   - LOW-VALUE REPRESENTATION IS 00 ####
CHANGE - NOTE REMOVED
ADD    - NOTE STATEMENT
ADD    - LINE
DELETE - LINE
CHANGE - BCL +(S) REPLACED BY EBCDIC +(S)
         TOTAL NUMBER OF LINES DELETED = ZZZZZZ
CHANGE - REDUNDANT PERIOD REMOVED FROM TEXT
CHANGE - QUOTES REMOVED FROM COPY TEXT-NAME
FLAG   - TEXT-NAME CONTAINS INVALID CHAR ####
CHANGE (DELETE) - TRACE DELETED FROM NON-PROCEDURE DIV
CHANGE - LITERAL MOVED TO AREA-B
FLAG   - LITERAL MUST START IN AREA-B ####
FLAG   - FILE-NAME NOT IN TABLE ####
CHANGE - 'L' IN COLUMN 7 TO '*'
FLAG   - LIBRARY FILE IS ALREADY CONVERTED ####
CHANGE - COPY FILE-NAME TO NEW NAME
FLAG   - LIBRARY FILE RPB EXCEEDS 9 - INVALID ####
CHANGE - WORD(S) REMOVED FROM DIVISION HEADER
FLAG   - ITEM MAY NOT BE USED IN ARITHMETIC
```

The following messages can appear in the Identification Division:

```
ADD    - CONVERSION AID COMMENT MESSAGE
DELETE - MONITOR STATEMENT
ADD    - PROGRAM-ID
CHANGE - QUOTES REMOVED FROM PROGRAM-NAME
CHANGE - NEW RESERVED WORD, XX APPENDED
CHANGE - PROGRAM-NAME REDUCED TO ONE WORD
CHANGE - INCORRECT ORDER - * PLACED IN COLUMN 7
CHANGE - IDENTIFICATION DIVISION ITEM CORRECTED
```

The following messages can appear in the Environment Division:

```
ADD    - CONFIGURATION SECTION HEADER
DELETE - INPUT-OUTPUT SECTION HEADER
ADD    - SOURCE-COMPUTER
CHANGE - INSERT SOURCE-COMPUTER NAME
```

```
ADD    - OBJECT-COMPUTER
CHANGE - COMPUTER NAME REDUCED TO ONE WORD
CHANGE - INSERT OBJECT-COMPUTER NAME
CHANGE - OBJECT-COMPUTER NAME
CHANGE - WITH SUPERVISOR CONTROL REMOVED
CHANGE - 'IS' REMOVED FROM MEMORY SIZE CLAUSE
CHANGE - MEMORY SIZE SPECIFIED AS CHARACTERS
CHANGE - TIME 60 REMOVED - REGISTER TIME
CHANGE - 'IS' INSERTED AFTER SEGMENT-LIMIT
FLAG   - CURRENCY SIGN MAY NOT BE L, /, OR = ####
CHANGE (DELETE) - REMOVE DEVICE IS MNEMONIC NAME
CHANGE - 'SPO' TO 'ODT'
FLAG   - ODT EQUATE MUST BE FIRST CLAUSE ####
FLAG   - CURRENCY MUST PRECEDE DECIMAL-PT ####
ADD    - INPUT-OUTPUT SECTION HEADER
FLAG   - EMPTY INPUT-OUTPUT SECTION INVALID ####
FLAG   - EMPTY FILE-CONTROL INVALID ####
CHANGE - SELECT CLAUSE MOVED TO AREA-B
CHANGE - DEVICE NAME IN ASSIGN CLAUSE
FLAG   - DATACOMM DEVICE NAME INVALID ####
FLAG   - DEVICE NAME LISTER INVALID ####
FLAG   - DEVICE NAME NOT FOUND IN TABLE ####
CHANGE (DELETE) - BY FILE REMOVED FROM SELECT
CHANGE (DELETE) - BY NN TO ATTRIBUTE SUBSYSTEM
CHANGE (DELETE) - WORK TO ATTRIBUTE UNIQUENAME
CHANGE (DELETE) - PROCESS OR ATTRIBUTE UNIQUENAME
CHANGE (DELETE) - MULTIPLE UNIT/REEL REMOVED
CHANGE (DELETE) - NO TRANSLATION REMOVED FROM SELECT
CHANGE (DELETE) - TRANSLATION NON-STANDARD REMOVED
CHANGE (DELETE) - FORM TO ATTRIBUTE FORMID
CHANGE (DELETE) - SAVE TO ATTRIBUTE PROTECTION
CHANGE - ALTERNATE AREA(S) TO (TOTAL) AREA(S)
CHANGE (DELETE) - FILE LIMIT(S) REMOVED FROM SELECT
CHANGE (DELETE) - WITH CHECK REMOVED FROM SELECT
CHANGE (DELETE) - PROCESSING MODE REMOVED FROM SELECT
CHANGE (DELETE) - SYMBOLIC KEY(S) REMOVED FROM SELECT
CHANGE (DELETE) - MULTIPLE FILE TO ATTRIBUTE FAMILY NAME
FLAG   - APPLY IS INVALID SORTER OPTION ####
CHANGE - RESERVE REMOVED FROM ASSIGN TO SORT
CHANGE - SHARED TO ATTRIBUTE SHARED
CHANGE - INSERT ACCESS MODE IS SEQUENTIAL
CHANGE - PERIOD DELETED FROM WITHIN I-O CONTROL
```

The following messages can appear in the Data Division:

```
CHANGE - FD TO SD FOR FILE ASSIGNED TO SORT
FLAG   - DATACOM FD IS INVALID ####
FLAG   - LISTER FD IS INVALID ####
FLAG   - SD REQUIRES 'SORT' IN SELECT ####
CHANGE - LEVEL INDICATOR MOVED TO AREA A
CHANGE - LEVEL NUMBER MOVED TO AREA A
CHANGE (DELETE) - RECORDING MODE REMOVED FROM FD
```

```
CHANGE - RECORDING MODE TO ATTRIBUTE EXTMODE
CHANGE (DELETE) - FILE CONTAINS REMOVED FROM NON-DISK
CHANGE (DELETE) - FILE CONTAINS TO AREASIZE & AREAS
CHANGE - LABEL RECORD TO ATTRIBUTE LABELKIND
CHANGE - VALUE OF ID TO ATTRIBUTE FILENAME
CHANGE - SAVE-FACTOR TO ATTRIBUTE SAVEFACTOR
CHANGE - 'ARE' TO 'IS' IN DATA RECORD CLAUSE
CHANGE - 'IS' TO 'ARE' IN DATA RECORDS CLAUSE
ADD    - FILE ATTRIBUTE FORMID
ADD    - FILE ATTRIBUTES AREASIZE, AREAS
ADD    - FILE ATTRIBUTE FAMILYNAME
ADD    - FILE ATTRIBUTE SUBSYSTEM
ADD    - FILE ATTRIBUTE PROTECTION
ADD    - FILE ATTRIBUTE UNIQUENAME
ADD    - FILE ATTRIBUTE SHARED
CHANGE - VALUE REMOVED FROM RECORD DESCRIPTION
FLAG   - RECORD SIZE MUST BE 390 CHARACTERS ####
CHANGE - ABBREVIATION(S)
ADD    - WORKING-STORAGE SECTION HEADER
ADD    - TALLY
ADD    - SWITCH DATA-NAME
CHANGE - MOD REMOVED FROM RECORD DESCRIPTION
FLAG   - PICTURE EXCEEDS 30 CHARACTERS    ####
CHANGE - I PLACED BEFORE INVALID PIC CHARACTER
CHANGE - INVALID PIC CHARACTER REPLACED BY /
FLAG   - INVALID PICTURE, SIGN MISPLACED ####
FLAG   - NUMERIC EDITED PIC, NUMERIC VALUE ####
CHANGE - SIGN REMOVED FROM VALUE
FLAG   - SIGNED VALUE REQUIRES SIGNED PIC ####
CHANGE (DELETE) - BZ / BLANK WHEN ZERO DELETED
CHANGE - DATA DECLARATION REARRANGED
CHANGE - LEADING J IN PICTURE TO S
FLAG   - J IS INVALID PICTURE CHARACTER  ####
CHANGE - LEADING K IN PICTURE TO S
CHANGE - DATE-COMPILED TO ZERO
FLAG   - OCCURS CLAUSE REQUIRES DEPENDING ####
CHANGE - FILE CLAUSE MOVED TO AREA B
FLAG   - SPLITTING OCCURS CLAUSE IS INVALID ####
FLAG   - REDEFINES MUST FOLLOW NAME ####
CHANGE - REDEFINES OBJECT TO ORIGINAL NAME
CHANGE - QUALIFICATION IN REDEFINES REMOVED
FLAG   - PIC USED WITH RENAMES IS INVALID ####
FLAG   - VALUE EXCEEDS PICTURE SIZE ####
CHANGE - OCCURS 0 TO OCCURS 1 TIMES
CHANGE - OCCURS INTEGER-2 GREATER INTEGER-1
CHANGE - INVALID DEPENDING CLAUSE REMOVED
FLAG   - KEY MUST BE BE OCCURS OR SUB ITEM ####
CHANGE - 01 LEVEL FILLER TO DATA-NAME
CHANGE - FD/SD/01 NAME REMOVED FROM COPY
FLAG   - COMPILER REQUIRES CORRECT NAME(S) ####
FLAG   - COMPILER REQUIRES CORRECT SIZE ####
CHANGE - 'ARE' TO 'IS' IN LABEL RECORD CLAUSE
```

```
CHANGE - 'IS' TO 'ARE' IN LABEL RECORDS CLAUSE
CHANGE - FILE-NAME IN LIBRARY FILE
FLAG   - COMPILER ALIGNS SYNC ITEM TO BYTE ####
```

The following messages can appear in the Procedure Division:

```
ADD    - SECTION-NAME AT PROCEDURE START
ADD    - SECTION-NAME AFTER END DECLARATIVES
CHANGE - OTHERWISE TO ELSE
CHANGE - MOVE TODAYS-DATE BECOMES ACCEPT. . .
FLAG   - TODAYS-DATE INVALID, SEE ACCEPT ####
CHANGE - MOVE DATE BECOMES ACCEPT...DAY
FLAG   - DATE INVALID, SEE ACCEPT VERB ####
CHANGE - MOVE TIME BECOMES ACCEPT...TIME
FLAG   - TIME FIELD IS 8 DIGITS HHMMSSSS####
CHANGE - MOVE TIME BECOMES ACCEPT...TIMER
FLAG   - TIME INVALID, SEE ACCEPT VERB ####
CHANGE - 'FROM ODT' REMOVED FROM ACCEPT
CHANGE - CORE-SIZE TO ATTTRIBUTE PRIMARYSTORAGE
FLAG   - CORE-SIZE INVALID ####
CHANGE - VERB MOVED FROM AREA-A TO AREA-B
CHANGE - BLANK INSERTED AFTER PERIOD
CHANGE - BLANK INSERTED BEFORE OPENING QUOTE
CHANGE - BLANK INSERTED AFTER CLOSING QUOTE
CHANGE - BLANK INSERTED BEFORE/AFTER OPERATOR
CHANGE - BLANK INSERTED AFTER COMMA(S)
CHANGE - BLANK INSERTED AFTER SEMICOLON(S)
FLAG   - IF RELATIVE INDEX, SPACE REQUIRED ####
CHANGE - TO INSERTED BEFORE LAST ADD OPERAND
FLAG   - SEGMENTATION MAY AFFECT 'AFTER' ####
CHANGE - CLOSE LOCK TO CLOSE SAVE
CHANGE - CLOSE LOCK CRUNCH TO CLOSE CRUNCH
CHANGE - CLOSE RELEASE CRUNCH TO CLOSE CRUNCH
CHANGE - CLOSE RELEASE TO CLOSE DISMISS
CHANGE - NO REWIND DELETED FROM CLOSE
CHANGE - CLOSE WITH LOCK TO FOR REMOVAL
CHANGE - WITH CRUNCH DELETED FROM CLOSE
CHANGE - REMOVE DELETED FROM CLOSE REEL
CHANGE - RELEASE DELETED FROM CLOSE REEL
CHANGE - PURGE DELETED FROM CLOSE REEL
CHANGE - UNIT DELETED FROM CLOSE STATEMENT
FLAG   - CLOSE REEL RELEASE NOT SPECIFIED ####
FLAG   - CLOSE REEL PURGE NOT SPECIFIED ####
CHANGE - CLOSE REMOVE TO CLOSE DISMISS
CHANGE - 'ALL' REMOVED FROM DISPLAY
CHANGE - 'UPON ODT' REMOVED FROM DISPLAY
FLAG   - MOD IS INVALID DIVIDE OPTION ####
CHANGE - ROUNDED REMOVED FROM REMAINDER
FLAG   - INVALID USE OF ELSE / OTHERWISE ####
CHANGE (DELETE) - NULL 'ELSE' DELETED
FLAG   - ENTER SYMBOLIC / COBOL INVALID ####
ADD    - INITIALIZE TALLY
```

```
CHANGE - EXAMINE VERB BECOMES INSPECT
CHANGE - NEXT STATEMENT BEGINS BELOW
CHANGE - 'FILL ... INTO' TO 'SEND ...'
CHANGE - 'FILL ... FROM' TO 'RECEIVE ...'
CHANGE - 'FILL FROM POLL' TO 'ACCEPT ...'
CHANGE - LOGICAL OPERATOR INSERTED
FLAG   - ABBREV CONDITION REQUIRES AND/OR ####
CHANGE - 'THEN' TO 'THAN' IN RELATION TEST
CHANGE - INTERROGATE TO CALL SYSTEM INTERROGATE
FLAG   - DISKPACK REQUIRES NEW SYNTAX ####
CHANGE - NUMERIC LITERAL(S) TO NON-NUMERIC
FLAG   - REFER TO NEW HEXADECIMAL RULES ####
CHANGE - OPEN INPUT-OUTPUT TO OPEN I-O
FLAG   - O-I IS INVALID OPEN OPTION ####
CHANGE - REVERSED DELETED FROM OPEN
CHANGE - NO REWIND DELETED FROM OPEN
CHANGE - INTO PHRASE REMOVED FROM READ
CHANGE - READ AT END TO READ INVALID KEY
CHANGE - READ INVALID KEY TO READ AT END
FLAG   - AT END/INVALID KEY REQUIRE IMPERATIVE ####
CHANGE - READ LOCK TO CALL INTRINSIC
CHANGE - READ LOCK-ONLY TO CALL INTRINSIC
CHANGE - READ LCOK-ONLY TO CALL INTRINSIC
CHANGE - READ LOCK-WAIT TO CALL INTRINSIC
FLAG   - RECORD-NAME NOT IN TABLE ####
FLAG   - SEARCH VARYING TWO NAMES INVALID ####
FLAG   - REFER TO NEW 'SEARCH WHEN' RULES ####
CHANGE - KEY CONVERSION REMOVED FROM SEEK
CHANGE - SEEK LOCK TO CALL INTRINSIC
CHANGE - SEEK LOCK-ONLY TO CALL INTRINSIC
CHANGE - SEEK LOCK-WAIT TO CALL INTRINSIC
FLAG   - CONVERT SEEK TO CALL INTRINSIC ####
FLAG   - USE COLLATING SEQUENCE/ALPHABET ####
CHANGE - MEMORY SIZE INSERTED IN SORT
CHANGE - SORT RELEASE TO SORT DISMISS
CHANGE - 'WITH' REMOVED FROM SORT STATEMENT
CHANGE - SIGN REMOVED FROM STOP LITERAL
CHANGE - 'ALL' REMOVED FROM STOP
FLAG   - REFER TO NEW 'SWITCH' RULES ####
CHANGE - SWITCH TO DATA-NAME, -XX APPENDED
CHANGE - TRACE 0 TO CALL SYSTEM TRACE-OFF
CHANGE - TRACE 1 TO CALL SYSTEM TRACE-ON
CHANGE - TRACE 2 TO CALL SYSTEM TRACE-ON
CHANGE - TRACE 3 TO CALL SYSTEM TRACE-ON
CHANGE - TRACE 20 TO CALL SYSTEM DUMP
CHANGE - TRACE 22 TO CALL SYSTEM DUMP
CHANGE - UNLOCK TO CALL INTRINSIC UNLOCK
CHANGE - 'AFTER' INSERTED AFTER USE VERB
CHANGE - USE...ON INPUT-OUTPUT TO USE...I-O
FLAG   - USE O-I IS INVALID OPTION ####
FLAG   - USE AT END OF PAGE IS INVALID ####
FLAG   - USE FOR KEY CONVERSION IS INVALID ####
```

```
FLAG   - USE LABEL PROCEDURE IS INVALID ####
FLAG   - CHECK TIMEOUT IN FILE STATUS ####
CHANGE - WRITE CHANNEL 1 TO WRITE PAGE
FLAG   - DEFINE CHANNEL IN SPECIAL-NAMES ####
CHANGE - ERROR/AUXILIARY PHRASE REMOVED
CHANGE - INSERT 'BEFORE ADVANCING 1 LINE'
FLAG   - RANDOM ACCESS REQ INVALID KEY/USE ####
CHANGE - WRITE LOCK TO CALL INTRINSIC
CHANGE - ZIP TO CALL SYSTEM ZIP USING
CHANGE - 'THEN' REMOVED FROM TEXT
FLAG   - LISTER CONSTRUCT IS INVALID ####
FLAG   - DATACOMM CONSTRUCT INVALID ####
FLAG   - SHARED FILES REQUIRE OVERFLOW/USE ####
FLAG   - CONVERT TO INTRINSIC DATE-COMPILED ####
CHANGE - 'EQUAL' REPLACED BY 'EQUAL'
FLAG   - INVALID USE OF NEXT SENTENCE ####
FLAG   - CONVERT SORTER CONSTRUCT TO 4A TYPE ####
```

Source Code Summary

A COBOL conversion source code summary follows the output listing. The following counts are listed:

```
SOURCE RECORDS READ
SOURCE RECORDS WRITTEN
SOURCE + LIBRARY RECORDS READ
SOURCE + LIBRARY RECORDS ADDED
SOURCE + LIBRARY RECORDS DELETED
SOURCE + LIBRARY RECORDS WRITTEN
CHANGE MESSAGES
FLAG MESSAGES
```

Glossary

The terms in this glossary are defined in accordance with their use in this document to describe COBOL. They might not have the same meaning for other languages. This glossary also reflects the complete contents of the American National Standard (ANSI-74), without the REPORT module and with Unisys extensions indicated. For this reason, everything appearing here might not yet be implemented in Unisys V Series COBOL.

A

abbreviated combined relation condition

The combined condition that results from the explicit omission of a common subject or a common subject and common relational operator in a consecutive sequence of relation conditions.

access mode

The manner in which records are to be operated upon within a file.

actual decimal point

The physical representation, consisting of either the period (.) or the comma (,), of the decimal point position in a data item.

actual key

A key whose contents identify a logical record in a sequential file.

alphabet-name

A user-defined word in the SPECIAL-NAMES paragraph of the Environment Division. The paragraph that assigns a name to a specific character set and/or collating sequence.

alphabetic character

A character that belongs to the following set: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, and the space.

alphanumeric character

A character in the computer's character set.

alternate record key

Other than the prime record key, a key whose contents identify a record within an indexed file.

arithmetic expression

An identifier or a numeric elementary item, a numeric literal, such identifiers and literals separated by arithmetic operators, two arithmetic expressions separated by an arithmetic operator, or an arithmetic expression enclosed in parentheses.

arithmetic operator

A single character or a fixed 2-character combination that belongs to the set: + (addition), − (subtraction), * (multiplication), / (division), and ** (exponentiation).

ascending key

A key, upon the values of which data is ordered, starting with the lowest value of the key up to the highest value of the key, in accordance with the rules for comparing data items.

assumed decimal point

A decimal point position that does not involve the existence of an actual character in a data item. The assumed decimal point has logical meaning but no physical representation.

at end condition

A condition caused during one of the following actions:

The execution of a READ statement for a sequentially accessed file

The execution of a RETURN statement when no next logical record exists for the associated sort or merge files

The execution of a SEARCH statement, when the search operation terminates without satisfying the condition specified in any of the associated WHEN phrases

B

block

A physical unit of data that is normally composed of one or more logical records. For mass storage files, a block can contain a portion of a logical record. The size of a block has no direct relationship to the size of the file within which the block is contained or to the size of the logical record or records that are either continued within the block or that overlap the block. The term is synonymous with physical record.

C

called program

A program that is the object of a CALL statement combined at object time with the calling program to produce a run unit.

calling program

A program that executes a CALL to another program.

character

The basic indivisible unit of the language.

character position

The amount of physical storage required to store a single standard data format character described as USAGE IS DISPLAY. Further characteristics of the physical storage are defined by the implementer.

character string

A sequence of contiguous characters that form a COBOL word, a literal, a PICTURE character string, or a comment entry.

class condition

The proposition, for which a truth value can be determined, that the content of an item is wholly alphabetic or is wholly numeric.

clause

An ordered set of consecutive COBOL character strings whose purpose is to specify an attribute of an entry.

COBOL character set

The complete COBOL character set consists of the following 52 characters: 0, 1 through 9 (digits), A through Z (letters), space (blank), + (plus sign), − (minus sign or hyphen), * (asterisk), / (slash, stroke, or virgule), = (equal sign), $ (currency sign), , (comma), ; (semicolon), . (period or decimal point)), " (quotation mark), ((left parenthesis),) (right parenthesis), > (greater than symbol), < (less than symbol), and @ (commercial at symbol)

COBOL word

See word.

collating sequence

The sequence in which the characters are acceptable in a computer for purposes of sorting, merging, and comparing.

column

A character position within a print line. The columns are incremented by and numbered from 1, starting at the leftmost character position of the print line and extending to the rightmost position of the print line.

combined condition

A condition that is the result of connecting two or more conditions with the AND or the OR logical operator.

comment-entry

In the Identification Division, an entry that can be any combination of characters from the computer character set.

comment line

A source program line represented by an asterisk in the indicator area of the line and any characters from the computer's character set in area A and area B of that line. The comment line serves only for documentation in a program. A special form of comment line represented by a slash or stroke (/) in the indicator area of the line and any characters from the computer's character set in area A and area B of that line causes page ejection prior to the printer's listing the comment.

compile time

The time at which a COBOL source program is translated by a COBOL compiler to a COBOL object program.

compiler directing statement

Beginning with a compiler directing verb, a statement that causes the compiler to take a specific action during compilation.

complex condition

A condition in which one or more logical operators act upon one or more conditions. *See also* negated simple condition, combined condition, negated combined condition.

computer-name

A system-name that identifies the computer upon which the program is to be compiled or run.

condition

The status of a program, for which a truth value can be determined at execution time. Where the term *condition* (condition-1, condition-2, ...) appears in these language specifications in or in reference to *condition* (condition-1, condition-2, ...) of a general format, the term is a conditional expression consisting of either a simple condition optionally parenthesized or a combined condition consisting of the syntactically correct combination of simple conditions, logical operators, and parentheses for which a truth value can be determined.

condition-name

(1) A user-defined word assigned to a specific value, set of values or range of values within the complete set of values that a conditional variable may possess. (2) The user-defined word assigned to a status of an implementer-defined switch or device.

condition-name condition

The proposition, for which a truth value can be determined, that the value of a conditional variable is a member of the set of values attributed to a condition-name associated with the conditional variable.

conditional expression

A simple condition or a complex condition specified in an IF, PERFORM, or SEARCH statement. *See* simple condition and complex condition.

conditional statement

A statement that specifies that the truth value of a condition is to be determined and that the subsequent action of the object program is dependent on this truth value.

conditional variable

A data item, one or more values of which has a condition name assigned to it.

configuration section

A section of the Environment Division, describing overall specifications of source and object computers.

connective

A reserved word that is used to associate a data name, paragraph name, condition name, or text name with its qualifier; link two or more operands written in a series; or form conditions (logical connectives). *See also* logical operator.

contiguous items

Items that are described by consecutive entries in the Data Division and that bear a definite hierarchical relation to each other.

counter

A data item used for storing numbers or number representations in a manner that permits these numbers to be increased or decreased by the value of another number or to be changed or reset to zero or to an arbitrary positive or negative value.

currency sign

The character *$* of the COBOL character set.

currency symbol

The character defined by the CURRENCY SIGN clause in the SPECIAL-NAMES paragraph. If no CURRENCY SIGN clause is present in a COBOL source program, the currency symbol is identical to the currency sign.

current record

The record that is available in the record area associated with the file.

current record pointer

A conceptual entity that is used in the selection of the next record.

D

data clause

A clause that appears in a data description entry in the Data Division and provides information describing a particular attribute of a data item.

data description entry

An entry in the Data Division that is composed of a level-number followed by a data-name, if required, and then followed by a set of data clauses, as required.

data item

A character or a set of contiguous characters (excluding literals in either case) defined as a unit of data by the COBOL program.

data-name

A user-defined word that names a data item described in a data description entry in the Data Division. When used in the general formats, *data name* represents a word that can neither be subscripted, indexed, nor qualified unless specifically permitted by the rules for that format.

debugging line

A debugging line is any line with *D* in the indicator area of the line.

debugging section

A debugging section is a section that contains a USE FOR DEBUGGING statement.

declarative sentence

A compiler-directing sentence consisting of a single USE statement terminated by the separator period.

declaratives

A set of one or more special purpose sections, written at the beginning of the Procedure Division, the first of which is preceded by the keyword DECLARATIVES and the last of which is followed by the keywords END DECLARATIVES. A declarative is composed of a section header, followed first by a USE compiler directing sentence and then by a set of zero, one, or more associated paragraphs.

default option

The option that is implicitly selected unless one of the other options is explicitly indicated.

delimiter

A character, or a sequence of contiguous characters, that identifies the end of a string of characters and separates that string of characters from the following string. A delimiter is not part of the string of characters that it delimits.

descending key

A key upon the values of which data is ordered starting with the highest value of the key down to the lowest value of the key in accordance with the rules for comparing data items.

destination

The symbolic identification of the receiver of a transmission from a queue.

digit position

The amount of physical storage required to store a single digit. This amount can vary, depending on the usage of the data item describing the digit position. Further characteristics of the physical storage are defined by the implementer.

division

A set of zero or more sections or paragraphs, called the division body, that are formed and combined in accordance with a specific set of rules. There are four divisions in a COBOL program: Identification, Environment, Data, and Procedure.

division header

A combination of words followed by a period and a space that indicate the beginning of a division. The division headers are IDENTIFICATION DIVISION., ENVIRONMENT DIVISION., DATA DIVISION., and PROCEDURE DIVISION [USING data-name-1 [data-name-2] ...] .

dynamic access

An access mode in which specific logical records can be obtained from or placed into a mass storage file in a nonsequential manner and obtained from a file in a sequential manner, during the scope of the same OPEN statement. *See also* random access and sequential access.

E

editing character

A single character or a fixed two-character combination belonging to the following set: B (space), 0 (zero), + (plus), − (minus), CR (credit), DB (debit), Z (zero suppress), * (check protect), $ (currency sign), , (comma), . (period or decimal point), and / (slash, stroke, or virgule).

elementary item

A data item that is not logically subdivided.

end of procedure division

The physical position in a COBOL source program after which no further procedures appear.

entry

A descriptive set of consecutive clauses terminated by a period and written in the Identification Division, Environment Division, or Data Division of a COBOL source program.

environment clause

A clause that appears as part of an Environment Division entry.

execution time

See object time.

extend mode

The state of a file after the execution of an OPEN statement with the EXTEND phrase specified and before the execution of a CLOSE statement.

family-name

A user-defined word that names a pack on which a COBOL library file resides.

figurative constant

A compiler generated value referenced through the use of certain reserved words.

file

A collection of records.

file clause

A clause that appears as part of any of the following Data Division entries: File description (FD), Sort-merge file description (SD), and Communication description (CD).

FILE CONTROL

The name of an Environment Division paragraph in which the data files for a given source program are declared.

file description entry

An entry in the File Section of the Data Division that is composed of the level indicator FD, followed first by a file-name and then by a set of file clauses, as required.

file-name

A user-defined word that names a file described in a file description entry or a sort-merge file description entry within the File Section of the Data Division.

file organization

The permanent logical file structure established at the time a file is created.

file section

The section of the Data Division that contains file description entries and sort-merge file description entries together with their associated record descriptions.

format

A specific arrangement of a set of data.

G

group item

A named contiguous set of elementary or group items.

H

hexadecimal literal

A character string bounded by @ (commercial at) signs. The string of characters must consist of one or more characters chosen from the set of hexadecimal digits consisting of the digits 0 through 9, A, B, C, D, E, and F. The characters A through F are the hexadecimal digit representation for the decimal values 10 through 15.

high memory

The memory areas above 1MD. High memory is the last segment in the overall layout of a program. If high memory is not used, the program stack is the last segment. The size of high memory is limited to 5MD and is fixed once the program is compiled. The size cannot be changed during program execution. The size is determined by the total data size allocated into the high memory areas.

high order end

The leftmost character of a string of characters.

I

I-O CONTROL

The name of an Environment Division paragraph in which object program requirements for specific input/output techniques, the sharing of same areas by several data files, and the multiple file storage on a single input/output device are specified.

I-O mode

The state of a file after execution of an OPEN statement, with the I/O phrase specified, and before the execution of a CLOSE statement.

identifier

A data name followed as required by the syntactically correct combination of qualifiers, subscripts, and indices necessary to make unique reference to a data item.

imperative statement

A statement that begins with an imperative verb and specifies an unconditional action to be taken. An imperative statement can consist of a sequence of imperative statements.

implementer-name

A system name that refers to a particular feature available on that implementer's computing system.

index

A computer storage position or register, the contents of which represent the identification of a particular element in a table.

index data item

A data item in which the value associated with an index name can be stored in a form specified by the implementer.

index-name

A user-defined word that names an index associated with a specific table.

indexed data-name

An identifier that is composed of a data-name, followed by one or more index-names enclosed in parentheses.

indexed file

A file with indexed organization.

indexed organization

The permanent logical file structure in which each record is identified by the value of one or more keys within that record.

input file

A file that is opened in the input mode.

input mode

The state of a file after execution of an OPEN statement with the INPUT phrase specified and before the execution of a CLOSE statement.

input-output file

A file that is opened in the I-O mode.

Input-Output Section

The section of the Environment Division that names the files and the external media required by an object program and which provides information required for transmission and handling of data during execution of the object program.

input procedure

A set of statements that is executed each time a record is released to the sort file.

integer

A numeric literal or a numeric data item that does not include any character positions to the right of the assumed decimal point. Where the term *integer* appears in general formats, integer must not be a numeric data item and must not be signed or zero unless explicitly allowed by the rules of that format.

intermediate data item

A signed numeric data item provided by the implementer that contains the results developed in the course of an arithmetic operation prior to the final result being moved to the resultant identifier, if any.

invalid key condition

A condition at object time caused when a specific value of the key associated with an indexed or relative file is determined to be invalid.

K

key

(1) A data item that identifies the location of a record.
(2) A set of data items that serve to identify the ordering of data.

key of reference

The prime or alternate key currently being used to access records within an indexed file.

keyword

A reserved word whose presence is required when the format in which the word appears is used in a source program.

L

language-name

A system name that specifies a particular programming language.

level indicator

Two alphabetic characters that identify a specific type of file or a position in hierarchy.

level-number

A user-defined word that indicates the position of a data item in the hierarchical structure of a logical record or that indicates special properties of a data description entry. A level number is expressed as a 1- or 2-digit number. Level numbers in the range 1 through 49 indicate the position of the data item in the hierarchical structure of a logical record. Level numbers in the range 1 through 9 can be written either as a single digit or as a zero followed by a significant digit. Level numbers 66, 77, and 88 identify special properties of a data description entry.

library-name

A user-defined word that names a COBOL library that is to be used by the compiler for a given source program compilation.

library text

A sequence of character strings and/or separators in a COBOL library.

Linkage Section

The portion of a COBOL program that defines the data available to both the calling and called programs.

literal

A character string whose value is implied by the ordered set of characters comprising the string.

logical operator

The reserved words AND, OR, or NOT. In the formation of a condition, either or both of AND and OR can be used as logical connectives. NOT can be used for logical negation.

logical record

The most inclusive data item. The level number for a record is 01.

low order end

The rightmost character of a string of characters.

M

mass storage

A storage medium on which data can be organized and maintained in both a sequential and nonsequential manner.

mass storage control system (MSCS)

An input/output control system that directs or controls the processing of mass storage files.

mass storage file

A collection of records that is assigned to a mass storage medium.

MCS

See Message Control System.

merge file

An ordered collection of records to be merged by a MERGE statement. The merge file is created and can be used only by the merge function.

message

The data associated with an end of message indicator or an end of group indicator. *See* message indicators.

Message Control System (MCS)

A communication control system that supports the processing of messages.

message count

The count of the number of complete messages that exist in the designated queue of messages.

message indicators

The conceptual indicators EGI (end of group indicator), EMI (end of message indicator), and ESI (end of segment indicator). They serve to notify the MCS that a specific condition exists (end of group, end of message, end of segment). Within the hierarchy of EGI, EMI, and ESI, an EGI is conceptually equivalent to ESI, EMI, and EGI. An EMI is conceptually equivalent to ESI and EMI. Thus, a segment can be terminated by an ESI, EMI, or EGI. A message can be terminated by an EMI or EGI.

message segment

The data that forms a logical subdivision of a message normally associated with an end of segment indicator. *See* message indicators.

mnemonic-name

In the Environment Division, a user-defined word that is associated with a specified implementer name.

MSCS

See mass storage control system.

N

native character set

The implementer-defined character set associated with the computer for which object code is generated.

native collating sequence

The implementer-defined collating sequence associated with the native character set.

negated combined condition

The logical operator NOT immediately followed by a parenthesized combined condition.

negated simple condition

The logical operator NOT immediately followed by a simple condition.

next executable sentence

The next sentence to which control is to be transferred after execution of the current statement is complete.

next record

The record that logically follows the current record of a file.

noncontiguous items

In the Working-Storage and Linkage Sections, the elementary data items that bear no hierarchical relationship to other data items.

nonnumeric item

A data item whose description permits its contents to be composed of any combination of characters taken from the computer's character set. Certain categories of nonnumeric items can be formed from more restricted character sets.

nonnumeric literal

A character string bounded by quotation marks. The string of characters can include any character in the computer's character set. To represent a single quotation mark character within a nonnumeric literal, two contiguous quotation marks must be used.

numeric character

A character that belongs to the following set of digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

numeric item

A data item whose description restricts its contents to a value represented by characters chosen from the digits 0 through 9. If signed, the item can also contain a plus (+), a minus (−), or another representation of an operational sign.

numeric literal

A literal composed of one or more numeric characters that also can contain either a decimal point, algebraic sign, or both. The decimal point must not be the rightmost character. The algebraic sign, if present, must be the leftmost character.

O

OBJECT-COMPUTER

The name of an Environment Division paragraph in which the computer environment within which the object program is executed is described.

object of entry

A set of operands and reserved words, within a Data Division entry, that immediately follows the subject of the entry.

object program

A set or group of executable machine language instructions and other material designed to interact with data to provide problem solutions. In this context, an object program is generally the machine language result of the operation of a COBOL compiler on a source program. Where there is no danger of ambiguity, the word *program* alone may be used in place of the phrase *object program.*

object time

The time at which an object program is executed.

open mode

The state of a file after execution of an OPEN statement and before the execution of a CLOSE statement. The particular open mode is specified in the OPEN statement as either INPUT, OUTPUT, I/O, or EXTEND.

operand

The component that is operated upon. For the purposes of this publication, any lowercase word or words appearing in a statement or entry format can be considered to be an operand and, as such, is an implied reference to the data indicated by the operand.

operational sign

In association with a numeric data item or numeric literal, an algebraic sign to indicate whether the value of the data item or literal is positive or negative.

optional word

A reserved word included in a specific format only to improve the readability of the language and whose presence is optional to the user when the format in which the word appears is used in a source program.

output file

A file that is opened in either the output mode or extend mode.

output mode

The state of a file after execution of an OPEN statement with the OUTPUT or EXTEND phrase specified for that file and before the execution of a CLOSE statement for that file.

output procedure

A set of statements to which control is given during execution of a SORT statement after the sort function is completed or during execution of a MERGE statement after the merge function has selected the next record in merged order.

P

paragraph

(1) In the Procedure Division, a paragraph name followed by a period and a space and by zero or more sentences.
(2) In the Identification and Environment Divisions, a paragraph header followed by zero or more entries.

paragraph header

A reserved word, followed by a period and a space that indicate the beginning of a paragraph in the Identification and Environment Divisions. The permissible paragraph headers in the Identification Division are PROGRAM-ID., AUTHOR., INSTALLATION., DATE-WRITTEN., DATE-COMPILED., and SECURITY. The permissible paragraph headers in the Environment Division are SOURCE-COMPUTER., OBJECT-COMPUTER., SPECIAL-NAMES., FILE-CONTROL., and I-O-CONTROL.

paragraph name

A user-defined word that identifies and begins a paragraph in the Procedure Division.

phrase

An ordered set of one or more consecutive COBOL character strings that form a portion of a COBOL procedural statement or of a COBOL clause.

physical record

See block.

prime record key

A key whose contents uniquely identify a record within an indexed file.

procedure

A paragraph or group of logically successive paragraphs, or a section or group of logically successive sections within the Procedure Division.

procedure name

A user-defined word used to name a paragraph or section in the Procedure Division. It consists of a paragraph name (which can be qualified) or a section name.

program name

A user-defined word that identifies a COBOL source program.

pseudotext

A sequence of character strings and/or separators bounded by, but not including, pseudotext delimiters.

pseudotext delimiter

A set of two contiguous equal sign (=) characters used to delimit pseudotext.

punctuation character

A character that belongs to the following set: , (comma), ; (semicolon), . (period), " (quotation mark), ((left parenthesis),) (right parenthesis), (space), = (equals sign), @ (commercial at sign).

Q

qualified data name

An identifier composed of a data-name followed by one or more sets of either of the connectives OF or IN followed by a data-name qualifier.

qualifier

(1) A data-name that is used in a reference together with another data-name at a lower level in the same hierarchy.
(2) A section name that is used in a reference together with a paragraph name specified in that section.

queue

A logical collection of messages awaiting transmission or processing.

queue name

A symbolic name that indicates to the MCS the logical path by which a message or a portion of a completed message might be accessible in a queue.

quietpoint

A point in time when there are no user programs in transaction state. *See also* transaction state.

R

random access

An access mode in which the program-specified value of a key data item identifies the logical record that is obtained from, deleted from, or placed into a relative or indexed file.

record

See logical record.

record area

A storage area allocated for the purpose of processing the record described in a record description entry in the File Section.

record description

See record description entry.

record description entry

The total set of data description entries associated with a particular record.

record key

A key, either the prime record key or an alternate record key, whose contents identify a record within an indexed file.

record name

A user-defined word that names a record described in a record description entry in the Data Division.

reference format

A format that provides a standard method for describing COBOL source programs.

relation

See relational operator.

relation condition

The proposition, for which a truth value can be determined, that the value of an arithmetic expression or data item has a specific relationship to the value of another arithmetic expression or data item. *See* relational operator.

relational character

A character that belongs to the following set: > (greater than), < (less than), and = (equal to).

relational operator

A reserved word, relation character, group of consecutive reserved words, or group of consecutive reserved words and relation characters used to construct a relation condition. The following are the operators and their meanings:

Relational Operator	Meaning
IS [NOT] GREATER THAN	Greater than or not greater than
IS [NOT] >	
IS [NOT] LESS THAN	Less than or not less than
IS [NOT] <	
IS [NOT] EQUAL TO	Equal to or not equal to
IS [NOT] =	

relative file

A file with relative organization.

relative key

A key whose contents identify a logical record in a relative file.

relative organization

The permanent logical file structure in which each record is uniquely identified by an integer value greater than 0. The value specifies the record's logical ordinal position in the file.

reserved word

A COBOL word specified in the list of words that can be used in COBOL source programs, but that must not appear in the programs as user-defined words.

resultant identifier

A user-defined data item that is to contain the result of an arithmetic expression.

routine name

A user-defined word that identifies a procedure written in a language other than COBOL.

run unit

A set of one or more object programs that function, at object time, as a unit to provide problem solutions.

S

section

A set of zero or more paragraphs or entries, called a section body, the first of which is preceded by a section header. Each section consists of the section header and the related section body.

section header

A combination of words followed by a period and a space that indicates the beginning of a section in the Environment, Data, and Procedure Divisions. In the Environment and Data Divisions, a section header is composed of reserved words followed by a period and a space. The permissible section headers in the Environment Division are CONFIGURATION SECTION. and INPUT-OUTPUT SECTION. The permissible section headers in the Data Division are FILE SECTION., DATA-BASE SECTION., WORKING-STORAGE SECTION., and LINKAGE SECTION. In the Procedure Division, a section header is composed of a section name, followed by the reserved word SECTION, followed first by a segment number (optional) and then by a period and a space.

section name

A user-defined word that names a section in the Procedure Division.

segment number

A user-defined word that classifies sections in the Procedure Division for purposes of segmentation. Segment numbers can contain only the characters 0 through 9. A segment number can be expressed as a 1-, 2-, 3-, or 4- digit number.

sentence

A sequence of one or more statements, the last of which is terminated by a period followed by a space.

separator

A punctuation character used to delimit character strings.

sequential access

An access mode in which logical records are obtained from or placed into a file in a consecutive predecessor-to-successor logical record sequence determined by the order of the records in the file.

sequential file

A file with sequential organization.

sequential organization

The permanent logical file structure in which a record is identified by a predecessor-successor relationship established when the record is placed into the file.

sign condition

The proposition, for which a truth value can be determined, that the algebraic value of a data item or an arithmetic expression is either less than, greater than, or equal to zero.

simple condition

A single condition chosen from the set relation condition, class condition, condition name condition, switch status condition, sign condition, and (simple condition).

sort file

A collection of records to be sorted by a SORT statement. Only the sort function can create and use the sort file.

sort-merge file description entry

In the File Section of the Data Division, an entry that is composed of the level indicator SD followed by a file name and a set of file clauses, as required.

source

The symbolic identification of the originator of a transmission to a queue.

SOURCE-COMPUTER

The name of an Environment Division paragraph describing the computer environment within which the source program is compiled.

source program

Although a source program can be represented by other forms and symbols, in this document it always refers to a syntactically correct set of COBOL statements beginning with an Identification Division and ending with the end of the Procedure Division. In contexts where there is no danger of ambiguity, the word *program* alone can be used in place of the phrase *source program.*

special character

A character that belongs to the following set: + (plus sign), − (minus sign), * (asterisk), / (slash, stroke, or virgule), = (equal sign), $ (currency sign), , (comma), ; (semicolon), . (period or decimal point), " (quotation mark), ((left parenthesis),) (right parenthesis), > (greater than symbol), < (less than symbol), @ (commercial at sign)

special-character w ord

A reserved word that is an arithmetic operator or a relation character.

SPECIAL-NAMES

The name of an Environment Division paragraph in which implementer-names are related to user-specified mnemonic names.

special registers

The compiler generated storage areas whose primary use is to store information produced in conjunction with the user of specific COBOL features.

standard data format

The concept used in describing the characteristics of data in a COBOL Data Division. The characteristics are expressed in a form oriented to the appearance of the data on a printed page, rather than a form oriented to the manner in which data is stored internally in the computer or on a particular external medium.

statement

A syntactically valid combination of words and symbols written, beginning with a verb, in the Procedure Division .

sub-queue

A logical hierarchical division of a queue.

subject of entry

An operand or reserved word that appears immediately following the level indicator or the level number in a Data Division entry.

subprogram

See called program.

subscript

An integer whose value identifies a particular element in a table.

subscripted data-name

An identifier that is composed of a data-name followed by one or more subscripts enclosed in parentheses.

switch-status condition

The proposition, for which a truth value can be determined, that an implementer-defined switch, has been set to an *ON* or *OFF* status.

zsyncpoint

A quietpoint that occurs when a user program initiates an END-TRANSACTION SYNC request, a user program closes a database, or the DASDL-specified number of transactions per syncpoint has been reached. *See also* quietpoint, transaction, transaction state.

system-name

A COBOL word that is used to communicate with the operating environment.

T

table

A set of logically consecutive items of data defined in the Data Division by the OCCURS clause.

table element

A data item that belongs to the set of repeated items comprising a table.

terminal

The originator of a transmission to a queue or the receiver of a transmission from a queue.

text-name

A user-defined word that identifies library text.

text-word

Any character string or separator, except space, in a COBOL library or in pseudo-text.

transaction

A logical grouping of database functions that are viewed by the user program as a single indivisible unit to be processed.

transaction state

A task is in transaction state when it issues a BEGIN-TRANSACTION request. A task leaves transaction state when it issues an END-TRANSACTION request. BEGIN-TRANSACTION and END-TRANSACTION requests are used for recovery to ensure consistency.

truth value

The representation of the result of the evaluation of a condition in terms of one of two values: true or false.

U

unary operator

A plus (+) or a minus (−) sign that precedes a variable or a left parenthesis in an arithmetic expression and that has the effect of multiplying the expression of +1 or −1 respectively.

unit

A module of mass storage, the dimensions of which are determined by each implementer.

user-defined word

A COBOL word that must be supplied by the user to satisfy the format of a clause or statement.

V

variable

A data item whose value can be changed by execution of the object program. A variable used in an arithmetic expression must be a numeric elementary item.

verb

A word that expresses an action to be taken by a COBOL compiler or object program.

W

word

A character string of not more than 30 characters, that forms a user-defined word, a system-name, or a reserved word.

Working-Storage Section

The section of the Data Division that describes Working-Storage data items, composed either of noncontiguous items, or of Working-Storage records, or of both.

77-Level-Description-Entry

A data description entry that describes a noncontiguous data item preceded by the level-number 77.

Index

A

abbreviated combined relation condition
(*See* combined relation condition)
abnormal termination
effect of ALTER statement on, 6–81
repeated branching from PERFORM range, 6–106
ABNORMALSAVE option
PROTECTION file attribute, 7–16
ACCEPT construct
Conversion Aid changes, F–23, F–41
ACCEPT MESSAGE COUNT statement
function, 6–32
interpretation of responses from, 6–135
queue inquiry construct, 6–135
ACCEPT statement
format 1
discussion of data transfer, 6–30
transfers of data, 6–30
format 2, 6–31
discussion of data transfer, 6–31
format 3, 6–32
formats, 6–30
function, 6–30
using a mnemonic-name, 4–6
access
dynamic, 4–19
random, 4–19
sequential, 4–19
ACCESS MODE clause
default, 4–19
effect on file access, 4–19
indexed files, 4–20
relative files, 4–20
using, 8–3
access routines
creating for indexed files, 7–12
program size,effect on, 8–5
ACCESSMODE file attribute
explanation of record access, 7–8
function, 7–8
ACCESSMODE option
FORCEIO attribute, 7–13
ACCUMIOTIME task attribute
explanation, 7–21
ACCUMPROCTIME task attribute
explanation, 7–21
ACTUAL KEY clause
declaring files as shared, 8–23
effect of WRITE statement, 6–171
effect on WRITE statement, 6–170
port file, subfile index, 6–117
requirement for port files, 4–18
updating ACTUAL KEY data item, 6–115
using with port files, 4–22
ACTUAL KEY data items
record access with READ, 6–116
ADD statement
Conversion Aid changes, F–23
format 1
operands and the word TO, 6–33
summation of values, 6–33
syntax, 6–33
format 2, 6–33
operands and GIVING, 6–34
summation of values, 6–34
format 3, 6–33
summation of data items, 6–34
formats 1 and 2
required references for identifiers, 6–34
function, 6–33
list of common features, 6–28
addition operator (+), 6–13
address errors (*See* errors)
ADVANCING phrase
AFTER specified, 6–173
BEFORE specified, 6–173
PAGE specified, 6–173
restriction on specification, 6–174
WRITE statement, 6–172, 6–174

AFTER option
  in comparison operations with INSPECT, 6–86
  PERFORM statement, 6–107
    multidimensional tables, 6–111
ALL
  as hexadecimal literal, 2–13
  figurative constant
    restriction in STOP, 6–146
ALL phrase
  INSPECT statement
    replacing, 6–87
    tallying, 6–87
ALL PROCEDURES phrase
  USE statements, 6–161
allocation of space
  in Linkage Section, 5–73
alphabetic data
  method of editing, 5–54
  to define an item as, 5–49
alphabetic data items, 5–6
alphabet-name clause
  literal specification, 4–8
  rules, 4–8
  Special-Names paragraph, 4–7
  syntax rules, 4–8
  value of literal, 4–8
alphabet-names
  and character sets, 4–5
  and collating sequences, 4–5
  as user-defined COBOL words, 2–4
  CODE-SET clause, 4–7
  converting character codes, 5–22
  function of, 2–4
  PROGRAM COLLATING SEQUENCE clause, 4–7
alphanumeric data
  method of editing, 5–54
  to define an item as, 5–49
alphanumeric data items
  defining alignment, 5–8
  range of values, 2–14
  relationship to other types, 5–6
alphanumeric file attributes (*See also* file attributes, alphanumeric), 7–1
alphanumeric-edited data
  method of editing, 5–54
  to define an item as, 5–50
ALSO phrase
  effect in collating sequence, 4–9
ALTER statement
  Conversion Aid changes, F–24
  discussion of elements, 6–35
  effect on GO TO statement, 6–81
  execution of debugging section, 6–163
  format, 6–35
  in input procedure for SORT, 6–142
  limitations on referencing paragraphs, 6–6
  modifying GO TO statement, 6–35
  order of execution with GO TO, 6–81
ALTERNATE RECORD KEY clause
  in indexed files, 4–21
  requirements, 4–21
  using, 8–7
alternate record keys
  purpose, 8–7
AND operator
  explanation, 6–20
ANSI Report Writer
  Conversion Aid changes, F–38
ANSI reserved words
  Conversion Aid changes, F–37
ANSIFILE option
  effect on record headers, 9–11
  function and conditions, 9–11
area A
  importance in division headers, 1–11
  using with replacements in COPY statement, 6–70
area B
  entries, 1–11
AREA option
  for DISKPACK files, 4–16
areas
  redefinitions
    multi-level 01 entries, importance of, 5–45
AREAS file attribute
  default value, 7–8
  function, 7–8
  restrictions, 7–8
AREASIZE file attribute
  explanation, 7–8
  restrictions, 7–8, 7–9
arithmetic expressions
  arithmetic operators, 6–13
  assigning value to data items, 6–65
  comparisons involving index data items, 6–17
  comparisons of nonnumeric operands, 6–16
  COMPUTE statement, 6–65
  definition, 6–12

arithmetic expressions (continued)
evaluating exponentiation, 6–14
formation and evaluation, 6–13
formatting rules, 6–13
function of sign condition, 6–20
in numeric function, 6–25
intermediate data items, 6–14
order of evaluation, 6–13
example, 6–13
sign condition, 6–20
storing the value, 6–14
using identifiers and literals, 6–12
using parentheses, 6–13
using relational operators, 6–16
using unary operators, 6–12
arithmetic operands
hexadecimal literals
restrictions, 2–14
arithmetic operations
format considerations
truncation of data, 6–28
maximum digits for numeric literals, 6–12
resultant-identifier, 6–26
truncating after decimal point alignment, 6–26
arithmetic operators
as reserved words, 2–10
examples, 6–13
arithmetic statements
as conditional statements, 6–10
common features, 6–28
format considerations, 6–28
format considerations for multiple results, 6–28
format of data descriptions, 6–28
list, 6–28
multiple results and DIVIDE statement, 6–78
producing multiple results, 6–28
using in numeric function, 6–25
ARM
using RESULT-DESCRIPTOR intrinsic, 6–43
arrays
example of illegal redefinition, 5–61
examples of legal redefinition, 5–61
ASCENDING clause
in MERGE
order of records, 6–92
effect on sorted sequence of data items in SORT, 6–141
ASCENDING KEY clause
SEARCH ALL operations, 6–130
ASCII
alphabet-names in CODE-SET, 5–22
characters code set and collating sequence, 4–8
collating sequence, 4–8
tables, C–1
ASSIGN clause
assigning indexed files, 4–17
assigning relative files, 4–17
associating a file with a storage medium, 4–14
default, 4–17
format 2, 4–17
format 3, 4–17
hardware descriptions, 4–14
hardware names, 4–14
in RESERVE clause, 4–18
placement, 4–17
port files, 4–18
RESERVE clause, 4–17
AT END condition
and USE procedures in USE, 6–165
movement of records under RETURN statement, 6–122
READ statement, 6–115
actions occurring, 6–115
with RETURN statement, 6–22, 6–122
AT END phrase
READ statement, 6–112
specification in SEARCH, 6–128
ATTRIBUTE CHANGEEVENT
function in WAIT statement, 6–168
ATTRIBUTE INPUTEVENT
function in WAIT statement, 6–168
ATTRIBUTE OUTPUTEVENT
function in WAIT statement, 6–168
attributes
with VALUE OF clause, 5–32
AVAILABLE phrase
OPEN statement, 6–100

B

BACKUP option
function, 4–16
BACKUPKIND file attribute
function, 7–9

BACKUPPERMITTED file attribute
  DONTBACKUP option, 7–9
  DONTCARE option, 7–9
  function, 7–9
  results of MUSTBACKUP, 7–9
BCI (*See* block control information)
BCL
  bit representation example, C–2
  column number specification, C–1
  Conversion Aid changes, F–37
  high-order bit specification, C–1
  language codes, table, C–8
  plus sign
  row number specification, C–1
  tables, C–1
  6-bit tape code, C–1
BCS (*See* Buffer Control Structure)
BEFORE phrase
  in comparison operations with INSPECT, 6–86
BINARY-DECIMAL option
  CALL INTRINSIC statement, 6–50
  description of data conversion, 6–50
  format requirements, 6–50
  truncation of digits, 6–50
binding
  advantages, list, 10–1
  calling a module, 10–1
  coding a program module, 10–1
  COMMON clause, 10–1
  creating modular programs considerations, 10–1
  declaring shared data items as COMMON, 10–2
  dollar options, using, 10–3
  error messages, B–18
  ICM dollar option, 10–2
  interlanguage without stand-alone binder, 6–36
  MAIN dollar option, 10–2
  main module declaring, 10–2
  module functions defining, 10–1
  passing information and parameters, 10–1
  program module entry points, 10–1
  recompiling module after changing, 10–2
  returning control to calling module, 10–2
  shared data areas defining, 10–1
bit representation
  BCL example, C–2
  EBCDIC example, C–1
  USASCII example, C–1
blank lines
  Conversion Aid changes, F–41
  definition, 1–12
BLANK WHEN ZERO clause
  elementary items, requirements, 5–37
  format, 5–37
  function, 5–37
  using with item whose PICTURE is numeric, 5–37
BLOCK CONTAINS clause
  format, 5–20
  function, 5–20
  using, 5–20
  variable length records, 5–20
block control information (BCI)
  effect on block size, 8–5, 8–9
  purpose, 8–9
  with relative files, 8–5
BLOCK file attribute
  function, 7–9
  interrogating, 7–9
block of shared disk
  locking, 6–46
block size, 7–10
  computing data size, 8–9
  effect of AREASIZE attribute, 7–8
  maximum for sort-merge files, 5–20
  relative files computing, 8–5 limitation, 8–5
  with variable length records, 5–20
blocked files
  buffer requirement, 4–18
  deblocking, 5–21
blocking factors
  performance considerations, 8–11
blocking technique
  for variable-length records, 7–10
blocking/deblocking
  function of COIxxx intrinsic, 9–8
blocks
  bad locating relative position, 6–90
BLOCKSIZE file attribute
  function, 7–9
  interrogating, 7–9
  limitation on modifications, 7–9

BLOCKSIZE file attribute (continued)
restrictions, 7–15
value, 7–9
BLOCKSTRUCTURE file attribute
FIXED and EXTERNAL values, 7–24
for workarea access files, 7–10
function, 7–10, 7–24
blue ink
purpose, 1–3
BNA network
using HOSTNAME attribute, 7–13
Boolean options
explanation, 9–9
ICMFAMILY, 9–15
ICMFILE, 9–15
LEV2IPC, 9–15
restoring previous value, 9–10
using $POP option, 9–10
using with stacks, 9–9
bounds checking
NOBOUNDS option, 9–19
using subscripts or indexes, 5–13
BPL
interlanguage binding
without stand-alone binder, 6–36
brackets (*See* special characters and notation)
branching
from PERFORM range, 6–106
procedure and IF statement, 6–82
BSRxxx intrinsic
function, 9–8
buffer areas
allocating with RESERVE clause, 4–18
Buffer Control Structure
overview, 8–12
buffer files
high data area, D–2
buffer size
HIGH MEMORY clause, 4–18
limitations, 4–18
buffer status word
assigning to buffer, 7–10
buffers (*See also* input/output areas), 4–18
allocating by host compiler, 8–16
allocating for indexed files, 8–8
allocating for shared indexed files, 8–14
allocating in nonshared indexed files, 8–12
alternate I/O
number of buffers assigned, 7–10
assigning buffer status word, 7–10
BLOCKSIZE attribute and size of block, 7–9
considerations for nonshared and shared indexed files, 8–16
definition, 4–18
determining length of data, 7–24
effect during file loads and writes, 8–18
effect of random updates, 8–18
effect on multiple keys, 8–18
effect with duplicate key values, 8–18
examining
FORCEIO attribute, 7–13
factors for allocating, 8–16
limitation on increasing, 7–10
random inquiry, 8–17
random update, 8–17
recommendations, 8–17
recommendations for multiple keys, 8–17
recommendations for one key, 8–17
relative files
allocation and size, 8–5
RESERVE nn AREAS clause, 8–16
sequential inquiry operations, 8–17
sequential update, 8–17
setting OUTPUTEVENT attribute, 7–27
BUFFERS file attribute
function, 7–10
limitations, 7–10
BY phrase
in PERFORM statement, 6–104
byte boundary
alignment of the subject data item in the computer, 5–66
of DISPLAY data item, 5–68
using an implicit 4-bit FILLER, 5–66
byte-alignment
of group items, 5–36

C

CALL INTRINSIC statement
  BINARY-DECIMAL option, 6–50
  DATE-COMPILED option, 6–43
  DATE-NOW option, 6–43
  DECIMAL-BINARY option, 6–50
  DISKFILEHEADER option, 6–47
  example, 6–40
  example of format 2, 6–39
  example of format 3, 6–39
  format 1, 6–39
  format 5, 6–40
  format 6, 6–40
  format 7, 6–40
  format 8, 6–41
  format 9, 6–41
  format 10, 6–42
  format 11, 6–42
  function, 6–39
  GETLABEL option, 6–52
  GETMCP option, 6–48
  GETPARM option, 6–43
  GETSWITCH option, 6–43
  JOBINFO option, 6–45
  MIX option, 6–43
  MIXID option, 6–45
  MIXNUM option, 6–45
  MIXTBL option, 6–43
  PROGINFO option, 6–44
  PUTLABEL option, 6–52
  READLOCK option, 6–46
  READLOCKONLY option, 6–46
  READLOCKWAIT option, 6–46
  RESULT-DESCRIPTOR option, 6–43
  SEEKLOCK option, 6–45
  SEEKLOCKONLY option, 6–46
  SETSWITCH option, 6–43
  SPOMESSAGE option, 6–47
  TIME-NOW option, 6–43
  TRANSLATE option, 6–51
  UNLOCK option, 6–46
  WRITELOCK option, 6–46
  ZIPSPO option, 6–49
CALL module
  documenting logical end-point, 6–79
CALL MODULE statement
  format, 6–53
  function, 6–53
  ICM-file-name, 6–53
  requirements for elements, 6–53
  transfer of control and EXIT MODULE, 6–53
  transferring control, 6–6
  transfers of control, 6–53
CALL statement
  called programs, 6–38
  compilation with $ICM set, 6–36
  data-names in USING phrase, 6–37
  elements, 6–36
  format, 6–36
  function, 6–36
  Linkage Section, 5–73
  logical relationship to cancelled program, 6–58
  ON OVERFLOW statement, 6–38
  operands in each USING phrase, 6–37
  USING phrase, 6–2
  within segmented program, 6–38
CALL SYSTEM statement
  DUMP option, 6–56
  format 1, 6–55
  format 2, 6–55
  format 3, 6–55
  function, 6–55
  INTERROGATE option, 6–57
  searching for file on resource pack, 6–57
  TRACE-OFF option, 6–56
  TRACE-ON option, 6–56
  ZIP option, 6–56
called programs
  cancelling, 6–58
  naming, 6–36
  returning control to calling program, 6–80
  status of, 6–38
calling and called programs
  using level 1 IPC, 9–15
  using level 2 IPC, 9–15
calling module
  returning control, 10–2
calling programs
  data-names in USING phrase, 6–37
  ON OVERFLOW statement, 6–38
  requirements of data descriptions, 6–2
CANCEL statement
  descriptions of elements, 6–58
  format, 6–58
  function, 6–58

CANCEL statement (continued)
  importance of setting $LEV21PC option, 6–58
  logical relationship to cancelled program, 6–58
CANDE
  creating library files, 6–71
CARD
  as primary file name, 9–5
card file
  as input for the COBOL compiler, 9–3
CARDIN
  & option control record, F–3
cardless compiles
  examples of compile statements, 9–5
  INSERT clause values, 9–4
  values for INSERT clause, 9–4
CARDOUT
  & option control record
    output options, F–3
CENSUS attribute
  effect of its value on INPUTEVENT, 7–25
  effect on OUTPUTEVENT attribute, 7–27
CHANGE ATTRIBUTE statement
  function, 6–59
  limitations, 6–59, 7–3
  syntax, 7–3
  syntax elements, 7–4
  using OF or IN clauses, 6–59
CHANGE statement
  examples, 6–59
  format, 6–59
  function, 6–59
  mnemonic file attribute, 6–59
  modifying file attributes, 7–3
  VALUE OF FILENAME declaration, 6–60
CHANGEDSUBFILE attribute
  function, 7–24
CHANGEEVENT attribute
  function, 7–24
  when set to *happened*, 7–24
CHANNEL clause
  example, 4–6
  Special-Names paragraph, 4–6
CHANNEL number
  with WRITE, 6–173
character code set
  function, 5–22
character codes
  converting from/to native codes, 5–22
character positions
  allocation
    in SORT procedures, 6–140
  vs. records in MERGE statement, 6–90
character representation
  USAGE clause, 5–67
character sets
  elements, 2–1
  example of translations between, 6–51
  for comments, 2–1
  for non-numeric literals, 2–1
character size
  physical
    specifying, 7–12
character string symbols
  PICTURE clause, 5–50
character strings
  COBOL words in, 2–3
  definition, 2–3
  delimiters, 2–3
  determining the category of an elementary item, 5–49
  linking, 2–2
  PICTURE, 2–3
  within pseudo text, 6–68
characteristics
  conceptual
    of files, 8–1
characters
  ETX
    size of receiving data-name, 6–30
  maximum on single-line with DISPLAY, 6–75
  placement of comma (,) in character string, 5–53
  restrictions on appearance of *P* and decimal point, 5–51
  scaling position
    required placement, 5–51
  tallying and/or replacing, 6–83
  transfers in STRING
    description of movement, 6–150
  when *V* becomes redundant, 5–52
CHARACTERS phrase
  INSPECT statement
    replacing, 6–87
    tallying, 6–87
  using in INSPECT statement, 6–85

CHECK option
  function, 9–11
  similarity to SEQCHECK, 9–20
class condition
  alphabetic test and use of DISPLAY, 6–18
  format, 6–18
  function, 6–18
  NOT and class tests, 6–18
  numeric test and use of DISPLAY or COMPUTATIONAL, 6–18
class tests
  explanation and example, 6–18
  position and representation of signs, 6–18
  restrictions on ALPHABETIC test, 6–18
  restrictions on using NUMERIC test, 6–18
clauses
  elements
    list, 1–3
  syntax
    definition, 1–3
CLEAR option
  function, 9–11
CLOSE operations
  effect on SUBFILEERROR attribute, 7–27
CLOSE statement
  and execution of debugging section, 6–161
  Conversion Aid changes, F–24, F–42
  following unsuccessful READ, 6–115
  format, 6–61
  function, 6–61
  limitation on execution, 6–62
  organization of files, 6–62
  relationship between files and close formats, 6–62
  relationship to OPEN statement, 6–99
  using DISMISS and RELEASE, 6–62
  using intervening OPEN statement, 6–62
  using REEL and UNIT, 6–62
  using REEL LOCK or UNIT LOCK, 6–62
  WITH NOT WAIT phrase, 6–62
  without REEL or UNIT phrases, 6–62
COBCIF
  internal and external file names, F–2
COBDIF
  internal and external file names, F–2
COBDOF
  internal and external file names, F–2
COBOL
  compiler
    advantages, 1–1
  debug module
    features, D–3
  interlanguage binding
    without stand-alone binder, 6–36
  language overview, 1–1
  program structure, 1–2
code listing, 9–12
code blocks
  including in compile listing, 9–17
code file
  generating, 9–3
  producing
    function of MAIN option, 9–17
code generation phase
  error messages, B–15
CODE option
  function, 9–12
code production
  discussion of efficiency and HIGH MEMORY, 5–43
code set differences
  USASCII X3.4–1963, C–1
  USASCII X3.4–1967, C–1
code set table
  USASCII X3.4–1963, C–2
  USASCII X3.4–1967, C–3
CODE-SET clause
  alphabet-names
    EBCDIC and ASCII, 5–22
  format, 5–22
  function, 5–22
  nonmass storage files, 5–22
  using literal phrases, 4–8
coding
  description of form, 1–6
  format considerations, 1–12
  program module, 10–1
  time saving tips, 6–71
  well structured
    example, 1–12
coding form
  considerations for layout, 1–12
  field definitions, 1–6

coding form (continued)
location of DECLARATIVES, 6-1
location of paragraph-name, 6-2
positions 73-80, 1-11
COFLTR
& option control record, F-3
coding rules, F-3
default values, F-2
inclusion of input and list option, F-3
list options, F-3
output options, F-3
purpose, F-2
remaining options, F-4
& Record errors, F-49
changes for
abbreviations in Data Division, F-19
ACCEPT and DISPLAY constructs, F-23
ACCEPT statement, F-41
ADD statement, F-23
ALTER statement, F-24
ANSI Report Writer reserved words, F-38
ANSI reserved words, F-37
BCL plus sign, F-37
blank lines, F-41
CLOSE statement, F-24, F-42
Configuration Section, F-8
COPY statement, F-41
CORE-SIZE register, F-34
data comm constructs, F-41
Data Division, F-15
DATE register, F-33
DISABLE statement, F-42
DISPLAY statement, F-42
DIVIDE verb, F-25
ELSE/OTHERWISE option, F-26
ENABLE, F-43
ENTER construct, F-26
Environment Division, F-7
EQUAL verb, F-28
EXAMINE format, F-26
figurative constants, F-40
File Section, F-16
FILE-CONTROL Paragraph, F-11
FILL Statement, F-28 F-43
Identification Division, F-7
IF statement, F-29
INPUT-OUTPUT Section, F-11
INTERROGATE statement, F-29, F-43
I-O CONTROL Paragraph, F-15
lister construct, F-46
literals, F-30
mass storage files, F-24
new reserved words, F-37
non-mass storage files, F-24
NOTE statement, F-37
OBJECT-COMPUTER Paragraph, F-8
OCCURS Clause, F-21
OPEN statement, F-30, F-43
periods, F-41
PICTURE Clause, F-20
Procedure Division, F-23
READ statement, F-30, F-44
reader-sorter construct, F-46
REDEFINES Clause, F-22
reference formats, F-41
RENAMES Clause, F-22
SEARCH statement, F-31
section headers, F-32
SEEK statement, F-32
separators, F-40
SORT statement, F-32
SOURCE-COMPUTER Paragraph, F-8
special registers, F-32
SPECIAL-NAMES Paragraph, F-10
STOP statement, F-34
switches, F-34
THEN separator, F-40
TIME register, F-33
TODAYS-DATE register, F-33
TRACE statement, F-35
Unisys reserved words, F-39
UNLOCK statement, F-35
USE statement, F-35
VALUE Clause, F-20
WAIT statement, F-44
Working-Storage Section, F-19
WRITE statement, F-36, F-44
WRITE-READ statement, F-44
WRITE-READ-TRANS, F-45
WRITE-TRANS-READ, F-45
ZIP statement, F-36
changes pertaining to more than one division, F-36
changes processed by Conversion Aid, F-7
COFLTR (continued)

comment record to prevent recycling output, F–1
compiler control record in input stream, F–3
control statements used to execute Conversion Aid, F–2
conversion from COBOL68 to COBOL74, F–1
errors on & option control record, F–3
exception conditions
equated hardware names, F–51
expansion of source coding, F–50
file names, F–50
input file validation, F–50
recognizing ending quote, F–51
record names, F–51
sequence numbers, F–51
types, F–49
exception conditons
& Record Errors, F–49
features, F–1
file equate statements, F–2
input character code set for file equate, F–2
invalid communications devices, F–12
listing for source code summary, F–58
messages list, F–53
NN option, F–12
old and new hardware names, F–11
output, F–52
output character code set for file equate, F–2
output messages
Data Division, F–54
Environment Division, F–53
Identification Division, F–53
more than one division, F–53
Procedure Division, F–56
reasons for flagging constructs, F–1
restrictions against rerunning a program, F–1
restrictions for compiling a program, F–1
stopping execution of Conversion Aid, F–1
valid hardware names, F–11
COIxxx intrinsic
function, 9–8
collating sequence
alphabet-name clause, 4–7, 4–8
arithmetic comparisons, 6–16
effects of ALSO phrase, 4–9
native, using, 4–4
ordinal number of characters, 4–8
THROUGH phrase, 4–9
value literal, 4–8
COLLATING SEQUENCE clause
in the SORT statement, 6–141
column number
EBCDIC, USASCII, BCL, C–1
combined conditions
conditions and logical operators, 6–22
format, 6–21
how they result, 6–21
using parentheses with logical operators, 6–22
valid combinations of NOT and OR, 6–22
combined relation conditions
abbreviated
conditions and guidelines, 6–23
format, 6–23
NOT, 6–23
using forms of abbreviation, 6–23
commas
function in syntax, 1–5
restriction on REPLACING, 6–70
comment entries
description, 2–14
comment lines
example, 1–8
in library text, interpreting, 6–68
in library text of COPY, 6–68
overview, 1–8
comments
date-compiled paragraph, 3–3
debugging lines, 4–2
separators in, 2–3
treatment of RIGHT and LEFT following SYNCHRONIZED, 5–66
COMMON clause
file description
format, 5–23
function, 5–23
format, 5–38
function, 5–38
required use in Data Division, 5–38
restrictions, 5–38

communication
synchronous
programs in multiprogramming mix, 6–132
RECEIVE statement, 6–118
with host, 7–25
comparison cycle
inspection process, 6–84
comparison operations
BEFORE and AFTER phrases, 6–86
CHARACTERS phrase, 6–86
in INSPECT, 6–86
punctuation and REPLACING phrase, 6–69
tallying or replacing literals, 6–85
text replacement with REPLACING phrase of COPY, 6–69
compilation
date and time, 6–42, 6–43
file names on dollar records, 9–4
terminating
ERRORLIMIT = nnn, 9–13
compilation date
format, 3–3
compilation file
function, 9–1
compilation speed
improving on large programs, 9–24
compile and go
syntax, 9–3
compile as comment lines
absence of WITH DEBUGGING MODE clause, D–3
compile debugging lines
WITH DEBUGGING MODE clause, D–3
COMPILE initiated program
if IX1 is non-zero at termination, 6–147
compile listing
DELETE and LISTDELETED, 9–12
including addresses and data block sizes, 9–17
placement of instruction addresses, 9–17
placement of subordinate data items, 9–17
when SYNTAX specified, 9–17
COMPILE statement
adding memory for workfile, 9–3
clearing IX1 at STOP RUN, 6–147
default options, 9–3
discussion, 9–2
examples, 9–5
format, 9–2
ICMs, treatment, 9–2
INSERT clause, 9–3
L option, 9–2
LIBRARY (LIB) option, 9–3
MEM option, 9–3
ON compiler-pack-name option, 9–2
ON object-pack-name option, 9–2
SAVE option, 9–3
SYNTAX option, 9–3
compiler
COBOL
advantages, 1–1
function, 1–2
general information, 9–1
documenting problems, 1–2
error and warning messages, B–2
error message list, B–17
expected input, 9–3
limits, 9–24
output listing, 9–22
table of I/O files and formats, 9–5
compiler address error
description of, B–1
compiler control options
overview, 1–11
compiler control record
conversion, F–3
if missing, 9–2
setting options, 9–20
compiler final code file
unreferenced data blocks, D–2
compiler intrinsics
BSRxxx, 9–8
COIxxx, 9–8
discussion, 9–8
DMPRDx, 9–8
INDxxx, 9–8
ISHxxx, 9–8
MANxxx, 9–8
RELxxx, 9–8
compiler operation
compilation file, 9–1
COMPILE statement, 9–2
dollar options, 9–9
program output, 9–22
sizes and limits, 9–24
source program, 9–7
strings in INSERT, 9–3

compiler output headings
TITLE option, 9–20
compiler size
sufficiency for compilations, 9–24
compiler stack
increasing size, 9–2
compiler stack overflow error
description of, B–1
compiler summary listing
SUMMARY option, 9–20
compiler work files
increasing size, 9–2
memory requirements, 9–24
compiler-directing sentences
definition, 6–10
compiler-directing statements
function, 6–10
compiler-directing verbs
COPY and USE, 6–10
compiles
cardless
examples of compile statements, 9–5
INSERT clause values, 9–4
table of INSERT clause values, 9–4
compile-time switch
debug module support, D–3
compiling
after changing, 10–2
compiling a program
restrictions before conversion, F–1
complex conditions
abbreviated combined relation conditions
format, 6–23
overview, 6–23
combined and negated combined conditions, 6–21
definition of truth values for, 6–20
discussion, 6–20
evaluation example, 6–25
formation, 6–20
logical operators and their meanings, 6–20
order of evaluation, 6–24
table of combined conditions and logical operators, 6–22
complex WAIT (*See* WAIT statement, complex)
COMPRESSION attribute
function, 7–24
COMPRESSIONPOSSIBLE attribute
function, 7–24
COMPUTATIONAL item
abbreviation, 5–67
definition, 5–67
range of values, 5–67
SIGN clause, 5–65
COMPUTE statement
checking for data overflow, 6–66
coding considerations, 6–65
computing value of arithmetic expression, 6–65
field lengths, 6–66
format, 6–65
function, 6–65
intermediate data items, 6–66
list of common features, 6–28
restrictions, 6–65
rules and explanations, 6–65
setting values with arithmetic expressions, 6–65
computer-name
as a system-name, 2–6
as system name, 4–2
definition, 2–6
function, 2–4
computer-name clause
definition, 4–2
condition
in IF statement, 6–82
condition statements
followed by conditionals in IF, 6–82
conditional expressions
ALPHABETIC class tests, 6–18
class condition, 6–18
format, 6–18
class tests, 6–18
combined conditions
format, 6–21
comparison of unsigned numeric operands, 6–16
comparisons and specified collating sequence, 6–16
comparisons involving index-names, 6–17
comparisons of nonnumeric operands, 6–16
comparisons of numeric operands, 6–16

conditional expressions (continued)
complex conditions, 6–20
combined and negated combined conditions, 6–21
negation of simple condition, 6–21
condition-name condition, 6–19
description of relation tests, 6–17
function, 6–15
negated simple conditions
format, 6–21
NUMERIC class tests, 6–18
relation condition
format, 6–15
function, 6–15
limitations, 6–15
truth values, 6–15
sign condition, 6–20
format, 6–20
simple condition
definition, 6–15
specifying, 6–15
switch-status condition, 6–19
format, 6–19
types, 6–15
using mnemonic attribute relation conditions, 6–17
conditional statements (*See* statements, conditional)
conditional variable
definition, 2–4
example, 5–39
testing using condition-name condition, 6–19
condition-name condition
format, 6–19
function, 6–19
rules for comparing, 6–19
testing a range of values, 6–19
when result of test is true, 6–19
condition-name entry
using VALUE clause, 5–69
condition-names
characteristics, 5–69
defining initial value, 5–69
definition, 5–39
examples, 5–39
general format, 5–39
linked with a conditional variable, level number 88, 5–45
qualification, 5–8
restrictions, 5–39
SEARCH statement, 6–130
testing, 4–7
Configuration Section
Conversion Aid changes, F–8
divisions, 4–2
function, 4–2
connectives
logical connectives, 2–8
qualifier connectives, 2–8
series connectives, 2–8
constants
figurative (*See* figurative constants)
constants and segment code
debugging aids, D–2
continuation lines
example, 1–9
overview, 1–9
using hyphen (-), 1–9
control
transfering with ALTER statement, 6–35
transferring with CALL statement, 6–36
transferring within Procedure Division, 6–81
transfers (*see* transfers of control)
control block
limitation on record storage, 8–10
purpose, 8–9
with relative files, 8–5
control block size
relative files
limitation, 8–5
control information
passing to the MCP, 6–49
control records
dollar options, 9–9
control statements
executing Conversion Aid program, F–2
conversion
& option control record
coding rules, F–3
default values, F–2
inclusion of input and list option, F–3
input options, F–3
list options, F–3
output options, F–3
purpose, F–2
remaining options, F–4
& Record errors, F–49

conversion (continued)
changes for
PICTURE Clause, F–20
changes for (*see* COFLTR, changes for)
changes processed by Conversion Aid, F–7
COBOL74 Conversion Aid Program (COFLTR), F–1
comment record to prevent recycling of output from Conversion Aid, F–1
compiler control record in input stream, F–3
control statements used to execute Conversion Aid, F–2
errors on & option control record, F–3
exception conditions, F–49
equated hardware names, F–51
expansion of source coding, F–50
file names, F–50
input file validation, F–50
recognizing ending quote, F–51
record names, F–51
sequence numbers, F–51
exception conditons & Record Errors, F–49
features of the COFLTR program, F–1
file equate statements, F–2
input character code set for file equate, F–2
invalid communications devices, F–12
list of messages. F–53
listing for source code summary, F–58
messages
COFLTR, F–53
NN option, F–12
old and new hardware names, F–11
output character code set for file equate, F–2
output from Conversion Aid, F–52
output messages
Data Division, F–54
Environment Division, F–53
Identification Division, F–53
more than one division, F–53
Procedure Division, F–56
reasons for flagging constructs, F–1
restrictions against rerunning a program, F–1
restrictions for compiling a program, F–1
stopping execution of Conversion Aid, F–1
valid hardware names, F–11
copied text
incorporating in source program, 6–72
restrictions, 6–72
COPY statement
coding examples, 6–70
Conversion Aid changes, F–41
creating libraries, 6–71
examples of REPLACING phrase elements, 6–68
format, 6–67
FROM clause, 6–68
function, 6–67
library feature, 6–71
library files
using, 6–72
logical equivalence to library-name, 6–67
ON clause, 6–67
processing, 6–67
pseudo text and REPLACING phrase, 6–68
REPLACING clause, 6–68
replacing entries, considerations, 6–70
required separators, 6–67
restrictions on text from COPY, 6–68
rules for library text, 6–67
using, 6–67
using COPYBEGIN and COPYEND, 6–72
using debugging lines, 6–68
using DISKPACK option, 6–68
using for library files, 9–7
using hexadecimal literals, 2–14
& option control record, F–5
CORE-SIZE register
Conversion Aid changes, F–34
core-to-core communication
in WAIT statement, 6–168
core-to-core option
MCP
further references to, 6–135
required setting, SEND, 6–133
using RECEIVE statement, 6–118
using SEND, 6–132
correction warning messages
suppressing, 9–12

CORRECTOK option
  function, 9–12
CORRECTSUPR option
  function, 9–12
CORRESPONDING phrase
  Data Division
    using with MOVE, 5–20
  in MOVE statement, 6–94
  keyword FILLER, 6–27
corruption of data
  changing labels, 7–15
COUNT IN phrase
  UNSTRING statement, 6–155, 6–157
CR
  as editing sign control symbol, 5–53
CRCR-INPUT clause
  WAIT statement, 6–168
CREATIONDATE file attribute
  function, 7–10
cross-reference listing
  XREF option, 9–21
CRUNCH option
  purpose, 6–71
CTLERR
  GIVING phrase of ZIPSPO, 6–50
CURRENCY SIGN clause
  using literal in PICTURE clause, 4–9
current date
  symbolic reporesentation, 6–43
current path
  saving, 9–20
CURRENTBLOCK file attribute
  function, 7–10
  using, 7–10
CURRENTRECORD attribute
  function, 7–25
CYLINDER option
  function, 4–16
CYLINDERMODE file attribute
  default value, 7–11
  function, 7–11

D

data
  alphabetic
    method of editing, 5–54
  alphanumeric
    method of editing, 5–54
  alphanumeric-edited
    method of editing, 5–54
  arrangement of
    using KEY IS phrase in
      OCCURS, 5–48
  categories, 5–1
  conversion
    binary to decimal, 6–50
    decimal to binary, 6–50
  corruption
    changing labels, 7–15
  defining initial condition-names, 5–69
  defining initial Working-Storage
    items, 5–69
  external representation, 5–22
  incompatible
    format considerations, 6–29
  see also transfers of data and DISPLAY
    statement, 6–74
  numeric
    method of editing, 5–54
  numeric-edited
    method of editing, 5–54
  organization
    using level numbers, 5–2
  overflow, 6–66
  SIXE ERROR condition, 6–66
  storage
    ACCEPT statement format 3, 6–32
  storage of translated, 6–51
  transfering
    ACCEPT statement, 6–30, 6–31
    maximum number of
      characters, 6–30
  transferring, low volume, 6–74
  transferring with DISPLAY, 6–74
  transfers of
    generation of DISPLAY LINES
      BCT, 6–74
  transfers
    see also transfers of data and
      DISPLAY statement, 6–74
  WRITE operation, purging, 6–176
data alignment
  rules defining, 5–8
  with JUSTIFIED clause, 5–8
data and code block identification
  debugging aids, D–1
data area
  in HIGH MEMORY
    limit on size, 5–42
data block size
  declaring for indexed file, 8–10

data blocks
including in compile listing, 9–17
referencing on compile listing, 9–17
data categories
defining an item as alphabetic, 5–49
defining an item as alphanumeric, 5–49
defining an item as alphanumeric-edited, 5–50
defining an item as numeric, 5–49
defining an item as numeric-edited, 5–50
in the PICTURE clause of the Data Division, 5–49
data classification
categories and classes, 5–6
effect editing method, 5–54
Data comm constructs
Conversion Aid changes, F–41
data conversions
elementary moves, in, 6–95
example of binary-decimal and decimal-binary, 6–51
internal representations
elementary moves, 6–95
data description entry
beginning with level number 01 or 77, 5–45
EBCDIC character set
using SIGN clause, 5–65
elementary items, 5–35
FD or SD
values of level numbers, 5–45
format 1, 5–33
format 2, 5–35
format 3, 5–35
function, 5–33
group items, 5–36
how level numbers are assigned, 5–35
inclusion of the COMMON clause, 5–38
indention considerations, 5–45
level number 77
function, 5–71
level number of USAGE clause, 5–68
nonspecified clauses, 5–35
numeric
if SIGN with a SEPARATE CHARACTER phrase applies to, 5–65
requirements when SIGN applies, 5–64
SIGN clause, 5–65
placement of data-name-1 or FILLER clause, 5–35
placement of REDEFINES clause, 5–35
range of numbers that can be assigned, 5–35
record descriptions, 5–33
required first element, 5–45
restriction on use of the OCCURS clause, 5–61
restrictions on PICTURE clause, 5–35
restrictions on REDEFINES clause, 5–61
using format 3 for condition-names, 5–39
using VALUE with OCCURS, 5–70
written order of clauses, 5–35
data descriptions
COBOL definition of all group items, 5–36
COMMON clause, 5–38
file-names in MERGE statement, 6–90
group, definition, 5–36
in Procedure Division, 6–2
OCCURS clause and table elements, 5–9
order of appearance of HIGH MEMORY clause, 5–42
subordinate
using SIGN clause, 5–64
with SYNCHRONIZED clause and an OCCURS clause, 5–66
with SYNCHRONIZED clause and an operational sign, 5–66
Data Division
BLANK WHEN ZERO clause, 5–37
BLOCK CONTAINS clause, 5–20
condition names, 2–4
Conversion Aid changes, F–15
Conversion Aid changes for abbreviations, F–19
Conversion Aid messages, F–54
data alignment, 5–8
DATA RECORDS clause, 5–24
data-names
use in qualification, 5–7
describing sort files, 6–140
error messages, B–3
file description entry, 5–16

Data Division (continued)
File Section, 5–16
definition, 5–1
HIGH MEMORY clause, 5–42
SYNCHRONIZED clause, 5–66
USAGE clause, 5–67
using PICTURE clause, 5–49
file-name description affecting RETURN, 6–122
file-name of MERGE statement, 6–90
format, 5–1
function, 1–2
general information, 5–1
HIGH MEMORY clause
requirements of specification, 5–42
level numbers
placement, 1–11
using, 5–2
LINAGE clause, 5–27
Linkage Section
definition, 5–1
function, 5–73
"not referenced" data blocks, 9–17
operational sign
describing position and mode of data, 5–64
record description
contents, 5–2
REDEFINES clause
function, 5–60
RENAMES clause, 5–62
sections, 1–2
SIGN clause, 5–64
sort or merge files, 4–14
structure, 5–1
syntax, 5–1
using LABEL RECORDS clause, 5–25
using OCCURS clause, 5–46
using tables, 5–9
using VALUE OF clause, 5–32
VALUE clause, 5–69
Working-Storage
definition, 5–1
discussion, 5–71
data exception entry
numeric
and the operational sign, 5–64
data items
ACTUAL KEY
accessing records with READ, 6–116
updating with READ, 6–115
allocating to the HIGH MEMORY areas of a program, 5–42
alphabetic, 5–6
alphanumeric, 5–6
range of values, 2–14
UNSTRING statement, 6–154
as index data item, 5–14
assigning value of arithmetic expression, 6–65
classes and categories
relationship, 5–6
conceptual
DATE field, 6–31
DAY field, 6–31
discussion, 6–31
TIME field, 6–31
TIMER field, 6–31
TODAYS-DATE field, 6–32
TODAYS-NAME field, 6–32
elementary
categories, 6–94
nonspecified clauses, 5–35
function of an identifier, 6–2
illegal moves, 6–95
in HIGH MEMORY
declaring as COMMON in other modules, 5–42
limitation on COMMON declaration, 5–42
way of referencing, 5–42
in LINAGE
how they are used, 5–28
index (*See* index data items)
instance when FILLER is generated for each occurrence, 5–66
instance when they are SYNCHRONIZED, 5–66
intermediate
checking overflow, 6–66
COMPUTE Statement, 6–66
definition and function, 6–14
limit of intermediate results, 6–34
limit to decimal digits, 6–14
situations that create, 6–14
truncation of operand, 6–14
using in MOVE statement, 6–93
using resultant-identifier, 6–14
internal representation, 5–67
linkage
order of processing, 6–2

data items (continued)
Linkage Section
Procedure Division
references, 5–73
monitoring
USE statement, 6–160
numeric, 5–6
division operation, 6–76
range of values, 2–14
subtracting, 6–151
positioning of data in a
receiving, 5–44
qualification, 5–7
RECEIVE statement, 6–118
receiving
when sending item is larger, 5–44
RECORD KEY, 4–21
deletions of records, 6–73
redefining another data item in HIGH
MEMORY, 5–42
redefining to table, 5–11
RELATIVE KEY
deletions of records, 6–73
SDF forms
using with HIGH MEMORY
clause, 5–42
sending
truncation of characters, 5–44
shared
declaring as COMMON, 10–2
STRING statement, 6–150
undefined values, 6–113
using as subscripts, 5–12
validity of moves between categories,
table, 6–94
when two or more modules refer to the
same, 5–38
data records
availability
use of REVERSED phrase, 6–102
how size is defined, 5–30
mass storage
accessing with SEEK, 6–131
multiple data-names, 5–24
shared areas, 5–24
specifying the size, 5–30
DATA RECORDS clause
coding example, 5–24
elements, 5–24
format, 5–24
function, 5–24
data references
logical record
structure, 5–2
qualifying, 5–7
using identifiers, 5–6
data size
consecutive occurrences of
characters, 5–50
storage considerations and *S*
symbol, 5–52
using $ in determining, 5–53
using / (slash) in determining, 5–53
using 0 (zero) in determining, 5–52
using astrisk (*) in determining, 5–53
using comma (,) in determining, 5–53
using editing sign control symbols in
determining, 5–53
using *9* in determining, 5–52
using *I* character string symbol, 5–51
using *V* in determining, 5–52
using *Z* in determining, 5–52
using period (.) in determining, 5–53
data transfers
AT END condition with
RETURN, 6–123
STRING functions, 6–150
STRING statement, 6–149
UNSTRING statement
effect of ALL phrase, 6–154
evaluating subscripting and
indexing, 6–154
rules, 6–156
treatment of characters, 6–156
updating DEBUG-ITEM
register, 6–162
using MOVE statement, 6–93
using ON OVERFLOW and POINTER
phrases with STRING, 6–150
data transmission
eligibility for in RECEIVE, 6–118
SEND statement
receiving programs eligible
for, 6–132
database
closing
and the DMSICM option, 9–12
name
DMSICM option, 9–12

data-names
as a subscript, 5–11
Data Division
using, 5–7
definition, 5–41
function, 2–5
general format for a, 5–41
in OFFSET function, 6–26
MERGE statement
rules, 6–90
qualification, 5–8
receiving
number of accepted characters, 6–30
specified in USING, 6–37
subscripted or indexed, 5–46
subscription when used as qualifier, 5–8
data-name-1
use in File-Control syntax, 4–13
data-name-2
use in File-Control syntax, 4–13
data-name-3
use in File-Control syntax, 4–13
date
symbolic representation, 6–43
DATE field
ACCEPT statement, 6–31
DATE register
Conversion Aid changes, F–33
DATE-COMPILED option
CALL INTRINSIC statement, 6–43
DATE-COMPILED paragraph
example, 3–3
format, 3–3
function, 3–3
DATE-NOW option
CALL INTRINSIC statement, 6–43
DAY field
ACCEPT statement, 6–31
DB
as editing sign control symbol, 5–53
DC keyboard command
input
what it displays and HIGH MEMORY, 5–42
DCP device
external file-name, 4–16
file assignment, 4–16
requirements for headers, 4–16
requirements for record size, 4–16
DCP files
WRITE functions, 6–176
deadlock
causes, 8–15
resolving, 8–15
debug module support
allocation of DEBUG-ITEM, D–3
compile-time switch, D–3
DEBUG-ITEM, D–3
features of COBOL language, D–3
WITH DEBUGGING MODE clause, D–3
DEBUG-CONTENTS data items
definition of and rules that apply to, 6–162
debugging
monitoring transfers of control, D–2
placement of debugging sections and USE statement, 6–159
USE FOR DEBUGGING, 6–158
USE statement
imperative verbs, 6–160
debugging aids
data and code block identification, D–1
high data area, D–2
MAP dollar option, D–2
object code file, D–2
object program memory layout, D–1
overlayed segments, D–2
reserved memory, D–1
segment code and constants, D–2
stack, D–2
debugging code
execution, controlling, D–2
debugging lines
as comment lines, 4–2
compiling, 4–2
example, 1–10
in library text, 6–68
overview, 1–10
program location, D–3
WITH DEBUGGING MODE clause, D–3
DEBUGGING MODE
reserving SW6, 4–7
debugging section
ALL REFERENCES OF phrase, 6–160
and USE statement, 6–159
automatic execution, discussion of, 6–159

debugging section (continued)
contents of DEBUG-SUB-1 through-3, 6–162
data items, discussion, 6–162
DEBUG-ITEM register, 6–161
GO TO statement, 6–163
implicit transfers of control, 6–163
PERFORM statement, 6–163
references in ALTER statement, 6–163
when control is transferred, 6–160
when executed, 6–161
when references in SORT or MERGE cause execution, 6–163
debugging sessions
using $TADS, 9–20
DEBUG-ITEM
allocation, D–3
debug module support, D–3
DEBUG-ITEM register
data items, contents, 6–162
DEBUG-CONTENTS data item, 6–162
example of implicit description, 6–161
function, 6–161, 6–162
function of DEBUG-NAME, 6–162
requirement of references to, 6–161
updating contents of, 6–162
DEBUG-LINE data items
function and contents of, 6–162
DEBUG-NAME
function, 6–162
separation of qualifiers, 6–162
decimal points
alignment with period (.), 5–53
assumed
using *V* in PICTURE, 5–52
placement in PICTURE clause, 5–52
using *P* in placing, 5–51
decimal representation
BINARY-DECIMAL intrinsic, 6–50
DECIMAL-BINARY option
CALL INTRINSIC statement, 6–50
data conversion, 6–50
numeric literals, requirements, 6–50
DECIMAL-POINT IS COMMA clause
effects on comma (,) and decimal point (.), 5–53
function, 4–9
declaratives
and port files, 6–1
beginning and ending, 6–1
definition, 6–1
grouping, 6–1
keywords
DECLARATIVES, 6–1
END DECLARATIVES, 6–1
placement, 6–1
required range of segment-numbers, 6–7
section
restriction on use of SORT, 6–138
USE statement, 6–1
DECLARATIVES header
placement of debugging section and USE statement, 6–159
DECLARATIVES keyword
Procedure Division
placement, 1–11
declaratives syntax, example, 6–3
default values
& option control records, F–2
DELETE option
function and conditions, 9–12
DELETE statement
as a conditional statement, 6–10
effect on record area, 6–73
format, 6–73
function, 6–73
general rules, 6–73
INVALID KEY condition, 6–73
INVALID KEY phrase
limitations, 6–73
updating value of FILE STATUS data item, 6–73
deleted images
listing in the output, 9–16
DELIMITED BY phrase
STRING statement, 6–149
UNSTRING statement, 6–154, 6–157
effect on data transfers, 6–156
DELIMITER IN phrase
UNSTRING statement, 6–155, 6–157
delimiters
for library programs, 6–71
for library texts, example, 6–72
pseudotext, 2–3
representation of in UNSTRING, 6–154
UNSTRING statement
use of figurative constants, 6–154
using ==, 2–3
using quotation marks, 2–2

delimiters (continued)
using the at (@) sign, 2–13
using with non-numeric literals, 2–11
DEMAND phrase
OPEN statement, 6–101
DENSITY file attribute
exceptions, 7–11
function, 7–11
options, 7–11
DESCENDING clause
in MERGE
order of records, 6–92
DESCENDING KEY clause
SEARCH ALL operations, 6–130
DESCENDING phrase
effect on sorted sequence of data items
in SORT, 6–141
DIALOGPROTOCOLLEVEL attribute
function, 7–25
digit offsets
placement on output listing, 9–22
dimensions of tables (*See* tables)
direct indexing (*See* indexing)
DIRECTION file attribute
explanation, 7–11
valid entries, 7–11
DISABLE statement
Conversion Aid changes, F–42
disjoint sets
forming with system-names, 2–6
using user-defined words, 2–4
disk
primary work medium, 4–17
disk files
file handling, 8–2
variable-length
COIxxx compiler intrinsic, 9–8
DISKFILEHEADER option
CALL INTRINSIC statement, 6–47
DISKIN
& option control record
input options, F–3
DISKOUT
& option control record
output options, F–3
diskpack
primary work medium, 4–17
diskpack files (*See* files, diskpack)
DISPLAY construct
Conversion Aid changes, F–23
DISPLAY data items
generating filler, 5–68
in record description, 5–68
internal representation, 5–68
DISPLAY LINES BCT
when generated, 6–74
DISPLAY statement
Conversion Aid changes, F–42
data transfer when DISPLAY contains
more than one operand, 6–74
display line length, 6–75
displaying messages on
ODT, 6–74
remote ODT, 6–74
elements, discussion, 6–74
format, 6–74
function, 6–74
maximum characters, 6–74
maximum characters on
single-line, 6–75
ON pack-name in displays, 6–75
UPON phrase, 6–74
using a mnemonic-name, 4–6
DISPLAY use of data item
and the SIGN clause, 5–65
DIVIDE statement
accuracy of REMAINDER data
item, 6–77
definition of the remainder, 6–77
effect with ROUNDED phrase, 6–77
format 1, 6–76
format 2, 6–76
format 3, 6–76
format 4, 6–77
format 5, 6–77
function, 6–76
GIVING phrase, 6–77
list of common features, 6–28
ON SIZE ERROR phrase, 6–78
REMAINDER phrase, 6–77
requirements of literals, 6–77
size error condition, 6–26
truncation of results, 6–77
DIVIDE verb
Conversion Aid changes, F–25
division
performing in program, 6–76
division headers
requirements, 1–11
division operator (/), 6–13
DMPRDx intrinsic
function, 9–8

DMS considerations
  LEV2IPC option, 9–15
DMSICM option
  database issues, 9–12
  function, 9–12
DMSII
  error messages, B–16
DMSII constructs
  and the DMSICM option, 9–12
DMSII files
  NORETURN option, 9–19
  status of indexed file after opened I/O, 8–10
  status of relative file after opening I/O, 8–5
documentation
  using DATA RECORDS, 5–24
dollar option MAP
  debugging aids, D–1
  function in object code file, D–2
dollar options
  ANSIFILE, 9–11
  as Boolean options, 9–9
  binding, using, 10–3
  CHECK, 9–11
  CLEAR, 9–11
  CODE, 9–12
  control records, 9–9
  CORRECTOK, 9–12
  CORRECTSUPR, 9–12
  DELETE, 9–12
  DMSICM, 9–12
  ERRORLIMIT = nnn, 9–13
  explanation of DOUBLE, 9–12
  explanation of FEDLEVEL, 9–13
  explanation of LIBRFAMILY, 9–16
  function and placement, 9–9
  ICM, 9–14
  ICMCODE, 9–15
  ICMFAMILY, 9–15
  ICMFILE, 9–15
  LEV2IPC, 9–15
  LIBR, 9–16
  library files, 9–7
  list, 9–11
  LIST, 9–16
  LIST$, 9–17
  list action indications, 9–9
  LISTDELETED, 9–16
  listing in output, 9–17
  LISTOMITTED, 9–16
  LISTP, 9–17
  MAIN, 9–17
  MAP, 9–17
  MCS, 9–17
  MERGE, 9–18
  NEW, 9–18
  NEWID, 9–19
  NOBOUNDS, 9–19
  NORETURN, 9–19
  OMIT, 9–19
  OPTIMIZE, 9–19
  PAGE, 9–19
  PAGESIZE, 9–19
  POP, 9–20
  SAVECP, 9–20
  SAVEREC, 9–20
  SEQ nnnnnn +nnnnnn, 9–20
  SEQCHECK, 9–20
  SEQUENCE, 9–20
  SUMMARY, 9–20
  TADS, 9–20
  TITLE "literal", 9–20
  user-defined
    declaration requirement, 9–10
    default values, 9–9
    maximum number allowed, 9–9
  using NOBOUNDS, 5–13
  VOID nnnnnn, 9–21
  WARNFATAL, 9–21
  WARNSUPR, 9–21
  XREF, 9–21
  $SET example, 9–10
dollar options (*See* compiler control options)
dollar record
  FEDLEVEL option not specified, 9–13
DONTBACKUP option
  BACKUPPERMITTED attribute, 7–9
DONTCARE option
  BACKUPPERMITTED attribute, 7–9
DOUBLE option
  function, 9–12
DOWN BY
  function in SET statement, 6–137
DS
  causes, 6–47
DUMP option
  CALL SYSTEM statement, 6–56
DUPLICATES phrase
  function, 4–21
  requirements, 4–21

dynamic access mode
  indexed files, 4–20
  NEXT phrase of READ, 6–112
  of indexed files
    key reference in READ, 6–116
  order of access
    in relative files, 8–4
  reading files, 6–112
  record retrieval for relative or indexed files with READ, 6–113
  relative files, 4–20

E

EBCDIC
  alphabet-names in CODE-SET, 5–22
  as default value of EXTMODE, 7–12
  bit representation example, C–1
  column number specification, C–1
  high-order bit specification, C–1
  representation and punch card, C–1
  row number specification, C–1
  table of interchange codes, C–4
  tables, C–1
EBCDIC character set
  using in source input files, 9–7
EC operation code
  generating, 6–42
EDITFL
  as primary file name for editor files, 9–5
editing
  data transfers with MOVE statement, 6–93
  fixed insertion, 5–54
  floating insertion, 5–55
    position of characters, 5–55
  insertion types, 5–53
  methods for various data categories table, 5–54
  PICTURE clause
    restriction on types of replacement, 5–54
  simple insertion, 5–54
  special insertion, 5–54
  suppression with replacement, 5–54
  using * and BLANK WHEN ZERO, 5–56
  using asterisks, 5–54
  using spaces, 5–54
  zero suppression, 5–56
editing rules
  PICTURE clause, 5–53
    suppression with replacement, 5–54
editing sign control symbols
  data items values and results, 5–55
  function in character string, 5–53
  using, 5–54
editing symbols
  using periods (.) in character string, 5–53
elementary data items
  categories, 6–94
  nonspecified clauses in data descriptions, 5–35
elementary item
  how a character string determines the category, 5–49
elementary items
  classification of data, 5–6
  definition, 5–2
  example, 5–36
  general characterstics and editing requirements, 5–49
  in data descriptions
    examples, 5–35
  non-contiguous
    definition, 5–71
  organization
    using level numbers, 5–2
  relationship with groups, 5–2
  specifying alignment, 5–66
  using BLANK WHEN ZERO clause, 5–37
  with USAGE clause, 5–68
elementary moves
  definition, 6–94
  valid, 6–95
ellipsis marks, 1–5
ELSE NEXT SENTENCE phrase
  omitting, 6–82
  using with IF statement, 6–82
ELSE phrase
  using in IF condition, 6–82
  when IF condition yields true result, 6–82
ELSE/OTHERWISE option
  Conversion Aid changes, F–26
ENABLE statement
  Conversion Aid changes, F–43

END DECLARATIVES keyword
  Procedure Division
    placement, 1–11
END option
  ON ERROR clause, 6–140
END statement
  causing program termination, 6–90
end-of-file action
  OPTIONAL attribute, 7–16
end-of-file calculations
  effect on LASTRECORD
      attribute, 7–15
END-OF-PAGE
  equivalence to EOP, 6–174
end-of-page condition
  with WRITE, 6–173
    LINAGE, 6–173
END-OF-PAGE phrase
  WRITE statement, 6–174
end-of-reel
  WRITE procedure, result, 6–174
end-of-unit
  WRITE procedure, result, 6–174
ENTER construct
  Conversion Aid changes, F–26
entry point
  program module, specifying, 10–1
Environment Division
  coding
    example, 4–33
  condition names, 2–4
  Configuration Section, 4–1
  Conversion Aid changes, F–7
  Conversion Aid messages, F–53
  error messages, B–2
  FILE-CONTROL paragraph
    effect of OPTIONAL on file
        close, 6–62
  function, 1–2
  Input-Output Section, 4–1
  overview, 4–1
  reference to mnemonic-name, in
      WRITE, 6–174
  sections, 1–2
  SPECIAL NAMES paragraph
    specifying mnemonic-name, 6–30
  SPECIAL-NAMES paragraph
    switch-status condition, 6–19
  specifying an alphabet-name for the
      COLLATING SEQUENCE clause
      in SORT, 6–141
  structure
    example, 4–1
  using segmentation, 6–5
EOF NO LABEL message
  when sort work file overflows, 6–140
EOP
  equivalence to END-OF-PAGE, 6–174
EQUAL verb
  Conversion Aid change, F–28
error condition branch
  displayed messages, 9–23
error conditions
  & option control record, F–6
  entries in FILE STATUS status key
      1, 4–22
  entries in FILE STATUS status key
      2, 4–23
  indexed files
    status of I/O, 4–27
  relative files
    status of I/O operations, 4–25
  shared indexed file, 8–22
  SIZE ERROR
    explanation, 6–26
    overflow and setting index-names in
        SET, 6–136
    when it applies, 6–26
  SIZE ERROR in exponentiation, 6–14
error detection
  during processing, 6–49
  monitoring transfers of control, D–2
error handling
  I/O
    USE statement (format 2), 6–158
  program
    USE statement, function, 6–158
error information
  gathering, using ZIPSPO, 6–50
error message
  table of fatal errors, 8–21
error messages
  binding phase, B–18
  code generation phase, B–15
  compiler, B–17
  compiler address error, B–1
  compiler stack overflow, B–1
  DMSII, B–16
  effect of $DOUBLE option, 9–12
  fatal, B–1
    indexed file handling, 4–30
    relative file handling, 4–26

error messages (continued)
fatal error encountered by the intrinsic, 8–21
indexed file error notification, 8–20
NIL pointer dereference, B–1
nonfatal, B–1, B–2
number preceding, 9–22
on output listing, 9–22
reserved words as user-defined words, A–1
shared indexed file, 8–22
syntax, B–12
table of status digit values for indexed files, 8–20
uninitialized pointer dereference, B–2
ZIPSPO, 6–50
ERRORLIMIT option
default value, 9–13
function, 9–13
errors
abnormal termination
ALTER statement with GO TO, 6–81
address
situations that can lead to, 6–106
detecting during processing, 6–49
effect of CORRECTOK dollar option, 9–12
effect of ERRORLIMIT = nnn option, 9–13
fatal
accessing non-present files, 7–16
logical attribute invalid, 7–1
fatal system
USE ON PROGRAM-FAULT statement, 6–165
handling for shared in DECLARATIVES, 6–47
handling I/O
USE ROUTINE, 6–117
indexed files
status keys 1 and 2, 4–30
locating position of bad blocks, 6–90
ON SIZE ERROR phrase in DIVIDE statement, 6–78
parity
using ON ERROR clause, 6–140
relative files
status keys 1 and 2, 4–26
repeated branching from PERFORM range, 6–106
run-time
and program output, 9–23
function of COIxxx intrinsic, 9–8
LEV2IPC option, 9–16
sequence
CHECK dollar option, 9–11
size error
on quotient, 6–78
on remainder, 6–78
size error condition, 5–13
syntax
entry in run log when COMPILE-initiated program terminates, 6–147
using ON ERROR clause, 6–90
using ON SIZE ERROR clause in COMPUTE, 6–66
ETX characters, 6–30
evaluating conditions
using IF statement, 6–82
evaluation
arithmetic expressions, 6–13
EVEN option
PARITY attribute, 7–16
EXAMINE format
Conversion Aid changes, F–26
REPLACING option, F–27
TALLYING option, syntax, F–26
example
brackets and braces, 1–4
COBOL syntax words, 1–3
coding structure, 1–12
comment lines, 1–8
continuation lines, 1–9
debugging lines, 1–10
punctuation, 1–5
examples
arrays, 5–61
BCL bit representation, C–2
calling ICMs, E–9
CHANNEL clause, 4–6
condition-names, 5–39
creating ICMs, E–8
DATA RECORDS clause, 5–24
DATE-COMPILED paragraph
dollar option MAP output, E–10
EBCDIC bit representation, C–1
ellipsis, 1–5
Environment Division coding, 4–33
Identification Division coding, 3–2

examples (continued)
indexed file, programmatic handling, E–5
MOVE statement, 6–93
numeric literals, 2–13
ODT clause, 4–6
passing tape label information, E–16
record locking, 8–15
RENAMES clause, 5–63
STOQUE parameter block, 6–133
STRING and UNSTRING, E–1
switch clause, 4–7
table initialization, 5–11
tables
one-dimensional, 5–9
three-dimensional, 5–10
tape label handling, E–16
USASCII bit representation, C–1
USE procedure
tape label handling, E–16
USE procedure to handle parity error, E–3
VALUE OF clause, 5–32
WAIT statement, 6–169
Working-Storage Section, 5–71
EXCEPTION condition
RECEIVE statement, 6–119
SEND statement, 6–133
USE statements, 6–164
exceptions
indexed file operation, 4–30
EXECUTE command
effect on clearing IX1 at STOP RUN, 6–147
execution
sequence, using PERFORM, 6–103
where it begins, 6–2
execution speed
using overlayable segments, 6–7
EXIT MODULE statement
binder
funtion, 10–2
effect on control in CALL MODULE, 6–53
placement in a paragraph, 6–80
EXIT PROGRAM statement
creating an ICM, 6–80
effect of $SET ICM record, 6–80
limitations, 6–80
placement in a paragraph, 6–80
EXIT statement
assigning procedure-name to point in program, 6–80
format 1, 6–79
format 2, 6–79
format 3, 6–79
function, 6–79
passing control to calling program, 6–80
purpose, 6–105
required placement in a paragraph, 6–80
returning control from called to calling program, 6–80
expiration date
tape files, 7–17
exponentiation
effect on field lengths in COMPUTE statement, 6–66
function of COIxxx intrinsic, 9–8
exponentiation operator (**), 6–13
EXTEND option
OPEN statement, 6–100
function, 6–100
extensions
notation in manual, 1–3
external media
representing data on, 5–22
EXTERNAL value
BLOCKSTRUCTURE port file attribute, 7–24
EXTMODE file attribute
function, 7–12
overriding, 7–12

F

family name
function, 2–5
FAMILYNAME file attribute
function, 7–12
fatal errors
accessing non-present files, 7–16
table, 8–21
FD level indicator
function, 5–19
FD (*See* file description entry)
federal level
warning message list, B–17

FEDLEVEL option
  function, 9–13
  if not specified, 9–13
  intended use, 9–13
  list of settings, 9–13
  warning messages, 9–13
FIB
  assigning value to file attributes, 5–32
  record length and ANSIFILE option, 9–11
  resident segment area, D–2
field lengths
  COMPUTE Statement, 6–66
FIFO mechanism
  benefits, in storage and retrieval functions, 6–135
figurative constants
  as literals
    in UNSTRING statement, 6–153
  Conversion Aid changes, F–40
  function, 2–8
  in DISPLAY statement, 6–74
  in INSPECT statement, 6–85
  in STRING statement, 6–148
  relationship to literals, 2–11
  required format, 2–8
  rules for representing a string, 2–10
  SPACE, 6–94
  types, 5–70
  using, 2–10
  using literals, 2–10
  VALUE clause, 5–69
  values and related reserved words, 2–9
  ZERO, 6–94
file access
  order
    file at remote host, 7–13
  SAME clauses, 4–31
  SECURITYUSE attribute, 7–18
  specifying who can access file, 7–17
  without SECURITYGUARD attribute, 7–17
file access methods
  file handling, 8–2
file assignment
  invoking the MERG: intrinsic, 4–17
  invoking the SORT: intrinsic, 4–17
file attribute identifier
  syntax, 7–3
file attributes
  ACCESSMODE, 7–8
  affect on system performance, 7–28
  alphanumeric, 7–1
    CHANGE ATTRIBUTE, 7–3
    requirements of TO clause, 6–59
  AREAS, 7–8
  AREASIZE, 7–8
  BACKUPKIND, 7–9
  BACKUPPERMITTED, 7–9
  BLOCK, 7–9
  BLOCKSIZE, 7–9
  BLOCKSTRUCTURE, 7–10
  Boolean, 7–2
  BUFFERS, 7–10
  categories, 7–1, 7–5
  checking for conflicts with operating system, 7–1
  CREATIONDATE, 7–10
  CURRENTBLOCK, 7–10
  CYLINDERMODE, 7–11
  DENSITY, 7–11
  DIRECTION, 7–11
  equating key files with data files, 4–17
  explanation of task attributes, 7–20
  EXTMODE, 7–12
  FAMILYNAME, 7–12
  FILENAME, 7–12
  FORCEIO, 7–13
  FORMID, 7–13
  general information, 7–1
  hardware restrictions, 7–5
  HOSTNAME, 7–13
  ignored, 7–1
  interrogating, 7–4
  KIND, 7–14
  LABELKIND, 7–14
  LASTRECORD, 7–15
  list, 7–5
  MAXRECSIZE, 7–15
  mnemonic, 7–2
    CHANGE ATTRIBUTE, 7–3
    comparisons involving, 6–17
  modifying subport, 6–59
  MYUSE, 7–15
  numeric, 7–2
    "1 relative", 7–2
    requirements of TO clause, 6–59
  OPEN, 7–16
  OPTIONAL, 7–16

file attributes (continued)
PARITY, 7–16
port and subport, 7–22
Procedure Division, modification, 7–3
PROTECTION, 7–16
redefinition, 6–59
references, 7–3
responsibility for logical correctness, 7–1
SAVEFACTOR, 7–17
SECURITYGUARD, 7–17
SECURITYTYPE, 7–17
SECURITYUSE, 7–17
SENSITIVEDATA, 7–18
SERIALNO, 7–18
SHARED, 7–18
SINGLEUNIT, 7–19
specification, 7–3
specifying subport, 7–4
SUBSYSTEM, 7–19
SYNCHRONIZED, 7–19
table, 7–5
that cannot be modified by CHANGE ATTRIBUTE, 6–59
UNIQUENAME, 7–20
UNITNO, 7–20
usage and restrictions, 6–59
VOLUMEINDEX, 7–20
file close
MERGE statement, options, 6–92
file closing
SORT statement
options for GIVING, 6–143
options for USING, 6–143
file description
COMMON clause
function, 5–23
DATA RECORDS clause
coding example, 5–24
FD
coding example, 5–24
file description entry
discussion, 5–19
if LINAGE specifies PAGE in a WRITE statement, 5–29
level indicator FD, 5–19
sort-merge entry (SD), 5–19
file description entry
Data Division, 4–13
format 1, 5–16
function, 5–16
sort-merge file
function, 5–16
file equate
input character code set, F–2
names used by Conversion Aid, F–2
output character code set, F–2
file functions
shared
exception-handling procedures, 8–24
file-access exceptions, 8–24
formats, 6–40
list, 8–23
providing, 6–39
setting SHARED, 8–23
file handling
overview, 8–1
file information block
manipulating or interrogating, 7–1
file information block (FIB)
assigning value to file attributes, 5–32
file information block (*See also* FIB), 7–1
file maintenance
sequential organization, 8–3
FILE MCP control instruction
and label equation, 9–6
file (*See* files)
file name
function, 2–5
file organization
media types, 8–2
SAME clauses, 4–31
file processing
OPEN statement, 6–98
file record areas
resident segment area, D–2
File Section
Conversion Aid changes, F–16
definition, 5–1
file description entry, 5–16
placement of a data-name in the, 5–41
placement of a FILLER keyword in the, 5–41
RECORD CONTAINS clause
function, 5–30
restriction on use of REDEFINES in level 01 entries, 5–60
using PICTURE clause, 5–49

FILE STATUS clause
description of status key 1, 4–22
function, 4–22
indexed files, 4–27
status keys 1 and 2, 4–28
relative files, 4–25
setting status key 1, 4–25
status key 2, 4–25
status keys 1 and 2, 4–26
results of I/O, 4–22
results of I/O operations, 4–23
sequential files, 4–22
status keys 1 and 2, 4–24
status key 2, 4–23
using, 8–3
FILE STATUS data item
declaration within program, 8–19
definition, 4–22
indexed files
status key 1, 4–27
status key 2, 4–27
position, 4–25
program error notification, 8–19
FILE STATUS variable
declaring files as shared, 8–23
file structure
using BLOCKSTRUCTURE attribute, 7–10
FILE-CONTROL paragraph
Conversion Aid changes, F–11
format for indexed files, 4–12
format for sequential files, 4–10
format for sort-merge files, 4–13
function, 4–10 8–3
port files
format, 4–13
file-id
UNIQUENAME attribute, 7–20
file-identifiers
example of changing, 9–6
external, 9–6
FILENAME file attribute
CHANGE ATTRIBUTE FAMILYNAME, 7–12
default filename, 7–12
function, 7–12
purpose, 7–12
VALUE OF FAMILYNAME, 7–12
file-names
external file-name, 4–16
from multiple file reel, in MERGE statement, 6–91
internal, 9–6
MULTIPLE FILE clause, 4–33
restriction on repetition in MERGE statement, 6–90
SAME clause
restrictions, 4–31
specifying shared or common, 5–23
files
access routines in program, 8–2
availability, OPEN statement, 6–98
backup issues, 4–16
binding programs, 10–1
blocked, 4–18
closing, 6–62
COBOL74 Conversion Aid Program (COFLTR), F–1
compiler work
effect of WRKP MCP option, 9–6
conceptual characteristics, 8–1
DCP
result header after READ, 6–117
WAIT ATTRIBUTE INPUTEVENT clause, 6–169
DCP device assignment, 4–16
defining
in Data Division, 5–1
deleting records, 6–73
determining status of I/O, 4–27
directing to printer or punch, 7–13
disk or diskpack
areas they can contain, 7–8
DISKPACK
using AREA option, 4–16
using CYLINDER option, 4–16
using SINGLE UNIT option, 4–16
dynamic access
identifying logically removed records, 6–73
equating to labels, 6–102
file names
exception conditions, F–50
ICM
compatibility in binding, 10–3
indexed
conditions upon completion of I/O, 4–27
indexed (*See* indexed files)
input
using OPTIONAL phrase, 4–14
input and output for compile, 9–4
input validation
exception conditions, F–50

files (continued)
internal and external names used by Conversion Aid, F–2
ISAM
declaring record keys, 8–13
ISC
WAIT ATTRIBUTE INPUTEVENT clause, 6–169
key (*See* key files)
library (*See* library files)
logical structure
ORGANIZATION clause, 4–19
maintenance
sequential organization, 8–3
mass storage (*See* mass storage files)
multiple file reel
restriction on file-names in SORT, 6–140
new
purging, 7–16
nonmass storage
using CODE-SET clause, 5–22
nonpresent
as INPUT files, 7–16
writing to, 7–16
not sort or merge
allocating storage areas, 4–32
opening, 6–99
opening in demand mode, 6–101
opening in flow mode, 6–101
organization method, restrictions, 8–2
output
backup device for, 7–9
random access mode in SEEK, 6–131
output from Conversion Aid, F–52
permanent
using OPTIONAL attribute, 7–16
physical aspects, 8–1
port (*See* port files)
preserving
using PROTECTION attribute, 7–16
primary symbolic and COMPILE INSERT clause, 9–3
printer
considerations when using the LINAGE clause, 5–29
printer backup
possible effect of LINAGE, 5–29
printer output
specifying backup device, 7–9
punch
backup device for, 7–9
random access
identifying logically removed records, 6–73
reader-sorter
OPEN DEMAND, 6–101
OPEN FLOW, 6–101
reading, 6–101
record area
effect of executing an OPEN, 6–99
record names
exception conditions, F–51
recovery
nonshared indexed files, 8–18
shared indexed files, 8–18
referenced in USE statement, 6–165
relative (*See* relative files)
removing
SENSITIVEDATA attribute, 7–18
sequenced
combining two or more, 6–89
sequential I/O module, 8–3
sequential (*See* sequential files)
shared (*See* shared files)
sharing
restrictions, 5–23
SORT
specifying TAPE or TAPES, 4–17
sort or merge
allocating storage areas, 4–32
sort-merge
format of File-Control paragraph, 4–13
maximum record sizes for various types of files, 5–31
named in SAME RECORD AREA clause, 6–121
specifying BLOCK CONTAINS clause, 5–20
using RELEASE to write records to, 6–121
specifying disposition, 6–61
unblocked
definition, 5–20
variable-length blocked
blocking technique, 7–10

files (continued)
workarea access
offset is greater than BLOCKSIZE, 7–10
writing additional records to end, 6–100
FILESTATE attribute
function, 7–25
list of mnemonic values, 7–25
FILESTATE changes
effect on CHANGEEVENT attribute, 7–24
FILL statement
Conversion Aid changes, F–28, F–43
FILLER clause
definition, 5–41
general format for the, 5–41
inclusion of filler digits, 5–66
keyword in CORRESPONDING phrase, 6–27
placement in a data description entry, 5–35
FILLER keyword
restrictions on explicitly referring to, 5–41
final code file
unreferenced data blocks, D–2
FIRST phrase
INSPECT statement
replacing, 6–88
fixed segments
definition, 6–5
FIXED value
of the BLOCKSTRUCTURE port file attribute, 7–24
FLOW phrase
OPEN statement, 6–101
footing area
definition, 5–28
FOOTING clause
default size of a logical page, 5–27
of LINAGE clause
in WRITE statement, 6–173
FOR phrase
STRING statement, 6–150
FORCEIO
causing performance degradation, 8–12
relative files
performance degradation, 8–6
using with nonshared indexed files, 8–12
FORCEIO attribute
purpose during write, 8–10
FORCEIO capability
with relative files, 8–6
FORCEIO file attribute
default value, 7–13
function, 7–13
importance of the access mode, 7–13
format
of manual, 1–3
format considerations
arithmetic statements, 6–28
data descriptions of operands, 6–28
decimal point alignment, 6–28
maximum size of operands, 6–28
incompatible data in Procedure Division, 6–29
multiple results in arithmetic statements, 6–28
overlapping operands, 6–28
producing multiple results, 6–28
rules for statements, 6–28
specific formats, 6–29
truncation of data, 6–28
FORMID file attribute
function, 7–13
using to print files, 7–13
value, 7–13
FORT77
interlanguage binding
without a stand-alone binder, 6–36
FORWARD option
DIRECTION attribute, 7–11
FROM clause
COPY statement, 6–68
FROM phrase
RELEASE statement, 6–121

G

GETLABEL option
CALL INTRINSIC statement, 6–52
GETMCP option
CALL INTRINSIC statement, 6–48
format requirements, 6–49
GIVING phrase, 6–49
USING phrase, 6–49
GETPARM option
CALL INTRINSIC statement, 6–43

GETSWITCH option
  CALL INTRINSIC statement, 6–43
GIVING
  ADD statement, 6–34
  in MULTIPLY statement, 6–97
GIVING clause
  of WAIT statement, 6–169
  WAIT statement, 6–168
GIVING phrase
  BINARY-DECIMAL option, 6–50
  GETLABEL option, 6–52
  in DIVIDE statement, 6–77
  in MERGE statement
    process, 6–91
  referencing files that are not
      open, 6–98
  SORT statement, 6–143
  with GETMCP option, 6–49
GO statement
  discussion of GO TO, 6–81
  effect on PERFORM return
      mechanism, 6–106
  format 1, 6–81
  format 2, 6–81
  function, 6–81
  identifier, definition, 6–81
  transfers of control, 6–81
GO TO statement
  execution of debugging section, 6–163
  imperative statements in
      SEARCH, 6–127
  in input procedure for SORT, 6–142
  modification by ALTER, 6–81
  order of execution for ALTER, 6–81
  restrictions, 6–81
  treatment if modified, 6–35
  using, 6–81
GO TO statements
  with consecutive imperative
      statements, 6–81
graphic representation
  EBCDIC, C–1
group
  definition, 5–36
group items
  alignment, 5–36
  COBOL's definition of all, 5–36
  COMPUTATIONAL
  creating with elementary items, 5–2
  effect on elementary items, 5–67
  example, 5–36
  organization
    using level numbers, 5–2
  specification of format 2 of the
      OCCURS clause, 5–47
  USAGE IS INDEX clause, 5–68
guard file
  and security, 7–17
GUARDED
  for shared indexed file, 8–14
GUARDED option
  SECURITYTYPE attribute, 7–17

H

hardware descriptions
  format with ASSIGN clause, 4–14
hardware devices
  MCP FILE statement, 9–6
hardware names
  format with ASSIGN clause, 4–14
hardware types
  KIND attribute, 7–14
headers
  description of paragraph, 1–11
  division, 1–11
hexadecimal literals
  as numeric literals, 2–13
  definition, 2–13
  restrictions, 2–14
  using in COPY statement, 2–14
high data area
  debugging aids, D–2
  Key files, D–2
HIGH MEMORY areas
  allocating a data item to the, 5–42
HIGH MEMORY clause
  declaring data items as COMMON in
      other modules, 5–42
  discussion of data items that redefine
      other data items, 5–42
  discussion of performance
      degradation, 5–43
  function in memory allocation, 4–18
  function in the Data Division, 5–42
  general format, 5–42
  recommendation for use, 5–43
  restriction on use with the $DMSICM
      option, 5–42
  restrictions, 5–42
    list, 5–42
  restrictions on the use, 5–42
  restrictions with ISAM files, 4–18

HIGH MEMORY clause (continued)
specification requirements, 5–42
using with buffers, 5–43
way of referencing data items in, 5–42
when $DMSICM is set, 4–18
high-order bit specification
EBCDIC, USASCII, BCL, C–1
HN keyboard command
and HOSTNAME initialization, 7–13
host
communication with, HOSTNAME attribute, 7–25
name of local, 7–26
host name
MYHOSTNAME attribute, 7–26
HOSTNAME
passing with GETMCP option, 6–48
HOSTNAME file attribute
files on remote host, 7–13
function, 7–13, 7–25
initializing hostname, 7–13
using null attribute, 7–23
when a BNA network is required, 7–13

I

ICM
creating with EXIT PROGRAM statement, 6–80
generating, 9–3
generating with $ICM and DMSICM, 9–12
ON object-pack-name option of COMPILE, 9–2
ICM dollar option
binding, 10–2
ICM files
compatibility
binding, 10–3
invoking, 3–2
ICM output file
DMSICM option, 9–12
ICMCODE option
function and conditions, 9–15
ICMFAMILY option
function and conditions, 9–15
ICMFILE option
function and conditions, 9–15
ICM-file-name
in CALL MODULE statement, 6–53
ICMs
calling, program example, E–9
creating, program example, E–8
ICM option, 9–14
ICMFAMILY option, 9–15
ICMFILE option, 9–15
invoking with CALL MODULE, 6–53
printing code from, 9–15
producing using MAIN option, 9–14
when $ICM is set, 6–53
Identification Division
coding example, 3–2
comments entries, 2–14
Conversion Aid changes, F–7
Conversion Aid messages, F–53
DATE-COMPILED paragraph
description, 3–3
function, 1–2
functions, 3–1
paragraphs
descriptions, 3–2
program-id paragraph, 3–2
reserved words and comments, 3–1
rules for formation, 3–1
syntax example, 3–1
identifier
function, 6–2
identifiers
and subscripts, 5–11
appearing in arithmetic expressions, 6–12
COMPUTE Statement, 6–65
definition, 5–6
in UNSTRING statement, 6–154
MULTIPLY statement, 6–97
PERFORM statement, 6–104
IF Block
purpose, 8–10
IF statement
as a conditional statement, 6–10
Conversion Aid changes, F–29
ELSE NEXT SENTENCE phrase, 6–82
ELSE phrase and passing of control, 6–82
format, 6–82
function, 6–82
nested, 6–82
paired IF and ELSE combinations, 6–82

IF statement (continued)
procedure branching or conditional statement, 6–82
testing, comparison with complex WAIT, 6–169
transfers of control, considerations, 6–82
using NEXT SENTENCE phrase, 6–82
when condition is false, 6–82
when condition is true, 6–82
images
from source and OMIT option, 9–19
imperative sentence
definition, 6–11
imperative statements
appearance of "imperative-statement" in the format of statements, 6–11
definition and function, 6–11
followed by conditionals in IF, 6–82
GO TO in consecutive sequence, 6–81
imperative verbs
list, 6–11
implementer names
relating to user-specified names, 4–5
Special-Names paragraph
purpose, 4–2
implementer-name
as a system-name, 2–6
definition, 2–7
IN
using for qualification, 6–3
using in qualification, 5–7
IN access to files
explanation, 7–18
indentifiers
in GO statement, 6–81
indention
using, 1–12
independent segments (*See* segments, independent)
independently compiled modules (ICMs), 6–53
index
minimum value, 5–13
subfile
updating ACTUAL KEY in READ, 6–117
subport (*See* subport index)
use in DEBUG-NAME, 6–162
INDEX BY phrase
defining index-name, 5–12
index data item
restrictions, 5–68
SEARCH or SET statement, 5–68
index data items
characteristics, 5–68
comparisons, 6–17
conversions, 5–68
operand in MOVE statement, 6–95
purpose, 5–15
restrictions on comparisons, 6–17
index name
function, 2–5
index validation
discussion, 5–13
index value
carrying as offset, 5–15
INDEXED BY clause
describing identifiers in SEARCH syntax, 6–127
index-names, SET statement, 6–136
INDEXED BY phrase
when it is required in the OCCURS clause, 5–48
indexed data name
placement when qualified, 5–13
indexed data-item
as data item, 5–14
indexed file
programmatic handling, example, E–5
indexed file handling routines
BSRxxx compiler intrinsic, 9–8
indexed file names
duplicate key file names, 7–12
indexed files
access methods, 4–20
access of nonshared files, 8–11
access routines, 8–2
access through ISAM, 8–7
accessing, 8–10
accessing records with READ, 6–116
accessing shared files, 8–13
allocating FIB and control information, 8–9
allocating input/output areas, 4–17
allocating key buffers, 8–8, 8–11
assigning to disk or diskpack, 4–17
block control information, 8–9
block size limitations, 8–10
blocking factor considerations, 8–11
buffer considerations, 8–16
computing total block size, 8–9

indexed files (continued)
conditions upon completion of I/O, 4–27
considerations of nonshared files, 8–11
control block size, 8–10
creating access routines, 7–12
data block size, 8–10
declaring, 8–7
FILE STATUS data item, 8–19
key, 8–9
outside of code file, 8–9
within a program, 8–9
determining if file is recoverable, 8–19
DYNAMIC access, 4–20
dynamic access mode, 8–7
effect of MCP on file recovery, 8–19
error conditions, 8–22
failure to CLOSE, 8–18
fatal error encountered by the intrinsic, 8–21
FILENAME attribute, 7–12
format of File-Control paragraph, 4–12
function of FORCEIO attribute, 7–13
identifying logically removed records, 6–73
implementation considerations, 8–9
INVALID KEY condition
REWRITE statement, 6–125
WRITE operation, 6–171
INVALID KEY phrase and REWRITE, 6–125
key data items, 8–7
key declarations, 8–11
key file blocking factor, 8–11
key file naming conventions, 8–9
key reference
unsuccessful READ, 6–115
limitation on record storage, 8–10
limits on keys, 8–10
listing number of blocks, 8–10
logical positioning for sequential retrieval of records, 6–144
logically removing records, 6–73
maximum open files for shared ISAM handler, 8–8
names
FILENAME attribute, 7–12
NEXT phrase of READ, 6–112
nonshared
access routines, 8–12
allocating buffers, 8–12, 8–16
buffer recommendations for multiple keys, 8–17
buffer recommendations for one key only, 8–17
consequence of FORCEIO, 8–12
declaring, 8–11
declaring alternate keys, 8–13
duplicate key values, 8–18
failure to CLOSE, 8–18
file loads and writes, 8–18
invoking FORCEIO, 8–12
key buffer recommendations, 8–17
opening, 8–12
random inquiry of buffers, 8–17
random update of buffers, 8–17
random updates and buffer size, 8–18
reading and writing modified block, 8–12
recovery, 8–18
resetting data or key block defaults, 8–12
REWRITES with multiple keys, 8–18
sequential update of buffers, 8–17
specifying key buffers, 8–12, 8–16
organization, 8–7
ORGANIZATION IS INDEXED, 4–19
organization of blocks, 8–9
outside of code file, 8–9
performance considerations, 8–11
prime record key, 8–7
program error notification, 8–19
ramdom access mode, 8–7
RANDOM access, 4–20
READ statement, KEY phrase, 6–116
reading, 8–7
reading new block, 8–10
reading, random access mode, 6–112
record access, 4–20
record availability, 6–113
recovery, 8–10, 8–18, 8–19
recovery of nonshared files, 8–18
recovery of shared files, 8–18
reducing key file I/Os, 8–11
references of specified KEY phrase in START, 6–144
resetting block defaults for FORCEIO, 8–12

indexed files (continued)
retrieving records, 8–7
sequential access mode, 8–7
sequentially accessed
records having duplicated values in alternate record key, 6–116
shared
allocating key buffers, 8–14
concurrent updates, 8–14
declaring, 8–13
declaring alternate keys, 8–13
declaring keys and descriptions, 8–13
file attributes, 8–14
hardware requirement, 8–13
implementation considerations, 8–13
ISAMDB, 8–13
ISHxxx, 8–13
limiting users, 8–13
list of error messages, 8–22
locking records, 8–15
memory requirement, 8–13
record locking example, 8–15
requirements, 8–13
resolving deadlocks, 8–15
security specification, 8–14
size of ISAM handler, 8–15
software requirement, 8–13
shared and nonshared mode, 8–8
shared files, 8–8
size limit for keys, 8–10
START comparisons, 6–145
START statement
establishing key of reference, 6–145
status of newly created file, 8–10
table of fatal errors, 8–21
table of status digit values, 8–20
using ALTERNATE RECORD KEY, 4–21
using SHARED attribute, 7–18
variable length records, 8–10
versus conventional files, 8–9
when the INVALID KEY condition exists, 6–125
within a program, 8–9
write beyond boundaries, result, 6–175
writing block, 8–10
indexed files (*see also* nonshared index files, shared indexed files), 8–9
indexed READ
physical I/O for file comparison, 8–12
indexed sequential access mode (*See* ISAM)
indexed sequential files
SHARED
ISHxxx compiler intrinsic, 9–8
indexed-names
order of occurrence, 5–12
restrictions on, 5–13
indexing
advantages over subscription, 5–15
direct
specification, 5–12
evaluation, in MOVE statement, 6–93
evaluation of
UNSTRING statement, 6–154
format, 5–13
losing advantage over subscripting, 5–15
multiplication for subscripts, 5–15
order of occurrence, 5–12
in RETURN and MOVE operations, 6–122
relative
determining resultant occurrence number, 5–12
effect on index-name, 5–15
specification, 5–12
using + and − operators, 5–12
syntax, 5–13
using INDEXED BY clause, 5–13
using multiple index-names, 5–12
using OCCURS clause, 5–12
versus subscripting, 5–14
index-names
adjustments with SEARCH statement, 6–126
associating with dimensions of tables, 6–127
comparisons involving, 5–15, 6–17
FROM phrase of PERFORM, 6–105, 6–107
incrementing
VARYING and INDEXED BY in SEARCH, 6–127
internal representation, 5–13
restrictions, SET statement, 6–136
using as table references, 5–12
VARYING or AFTER phrase of PERFORM, 6–105, 6–107

index-names (continued)
VARYING phrase with
SEARCH, 6–127
index-names in the OCCURS clause
requirements, 5–48
indicator area
continuation line, 1–9
debugging line, 1–10
function, 1–8
INDxxx intrinsic
explanation, 9–8
input files
accepted input media, 9–7
characteristics, 9–7
library files, 9–7
merging, 9–7
optional, 6–101
primary and secondary, 9–7
representation in COMPILE INSERT
clause, 9–3
symbolic
using MERGE option, 9–18
using dollar options, 9–9
using OPTIONAL phrase, 4–14
input messages
specifying number that can be
queued, 7–26
input options
& option control record, F–3
INPUT phrase
OPEN statement, checking label
records, 6–102
input procedure
GO TO
and PERFORM statements, 6–142
in SORT operations
restriction on use of ALTER
SORT operations, 6–142
input source files
blocking factors, 9–5
INPUTEVENT attribute
as a port attribute, 7–25
as a subport attribute, 7–25
equivalence to READ-OK in
WAIT, 7–25
input/output areas
allocating for data files, 4–17
specifying, 4–18
input/output control
Environment Division
overview, 4–1
input/output error handling
USE statement, 6–158
INPUT-OUTPUT section
Conversion Aid changes, F–11
Input-Output Section
File-Control
function, 4–10
File-Control paragraph
function, 4–10
I-O-Control paragraph
function, 4–10, 4–31
required elements, 4–10
INSERT clause
assigned values, 9–4
COMPILE statement, 9–3
values for cardless compiles, 9–4
values specified, 9–5
INSERT command
setting switches, 4–7
insert parameter
moving the value, 6–43
INSPECT statement
effect of TALLYING option, 6–85
elements, 6–84
examples
REPLACING function, 6–88
examples of TALLYING count, 6–88
format 1, 6–83
format 2, 6–83
format 3, 6–84
function, 6–83
inspection process, 6–84
matching and tallying, 6–85
references as USAGE IS
DISPLAY, 6–85
rules for replacements, 6–87
tallying
rules, 6–87
using CHARACTERS phrase, 6–85
inspection process
where it begins on data item, 6–84
instruction addresses
placement on output listing, 9–22
interchange codes
EBCDIC, C–4
interlanguage binding
without stand-alone binder, 6–36
intermediate data item (*See* data items,
intermediate)

inter-program communication
  data descriptions, 5–73
  describing data for, 5–1
  items in Linkage Section, 5–73
  LEV2IPC option, 9–15
  level II, independent execution of program containing EXIT PROGRAM, 6–80
  required data clauses, 5–73
  restriction on data items declared with HIGH MEMORY clause, 5–42
INTERROGATE option
  CALL SYSTEM statement, 6–57
  using DISKPACK clause, 6–57
INTERROGATE statement
  Conversion Aid changes, F–29, F–43
intersystem control
  non-BNA mode implementation, 4–16
INTNAME attribute
  as default value for TITLE, 7–27
  function, 7–26
INTO phrase
  of RETURN statement, 6–122
INTRINSIC ABORT call
  clearing IX1 to zero after program termination, 6–147
intrinsics
  discussion of "MERG:", 4–17
  discussion of "SORT:", 4–17
  system dependent
    formats, 6–39
    providing, 6–39
  system dependent (*See also* CALL INTRINSIC statement), 6–39
invalid key
  exception-handling procedures, 8–24
INVALID KEY condition
  during DELETE operation, 6–73
  indexed files
    READ, results in an, 6–116
  relative files
    READ statement, 6–116
  sequential files
    READ statement results in, 6–116
  START statement, 6–145
  with REWRITE statement, 6–125
  WRITE operation
    effect, 6–172
    indexed file, 6–171
    relative file, 6–171
    sequential file, 6–171
INVALID KEY phrase
  in WRITE statement, 6–170
  READ statement, 6–112
  restriction on using with DELETE, 6–73
  REWRITE statement, 6–125
  START statement, 6–144
IO access to files
  explanation, 7–18
I/O buffers
  alternate
    number of buffers assigned, 7–10
I/O errors
  handling
    USE ROUTINE, 6–117
I/O file
  releasing logical records, 6–170
I/O files and formats
  table of, 9–5
I/O operations
  conditions upon completion, 4–22
  effect on SUBFILEERROR attribute, 7–27
  facilitating overlap with processor operations, with SEEK, 6–131
  indexed files
    status key 1, 4–27
    status key 2, 4–27
  relative files
    conditions upon completion, 4–25
    values of status key 2, 4–25
  results in status key 2, 4–23
  sequential files
    FILE STATUS data items, 4–22
  status key 1, 4–22
  using OPEN statement, 6–98
I/O procedures
  associated with SORT or MERGE, 6–90
I/O resource time
  obtaining amount, 7–21
I/O statements
  causing AT END condition, 6–115
  conversions, 5–68
  indexed data items, 5–68
  OPEN statement, 6–99
  restriction on execution, 5–19

I-O CONTROL paragraph
  Conversion Aid changes, F–15
  format, 4–31
  function, 4–31
  MULTIPLE FILE clause, 4–33
  SAME clause, 4–31
IPC calls
  level 2
    MANxxx intrinsic in handling, 9–8
IPC calls with parameters
  restriction on passing data, 5–42
IS EQUAL TO relational operator
  in START statement, 6–144
ISAM
  access of indexed files, 8–8
  accessing indexed files, 8–7
  setting SHARED attribute, 8–8
ISAM files
  using HIGH MEMORY clause, 4–18
ISAMDB
  using, 8–13
ISC control commands
  in message header, 4–16
ISHxxx
  using, 8–13
ISHxxx intrinsic
  function, 9–8
item-names
  non-contiguous
    in Linkage Section, 5–73
IWE II
  COBOL debugging sessions and
    $TADS, 9–20
IX1
  clearing to zero, 6–147
  compiler function when there are not
    syntax errors, 6–147

J

job environment
  getting information on, 6–45
JOBINFO option
  CALL INTRINSIC statement, 6–45
jobs
  getting information on, 6–44
JUSTIFIED clause
  abbreviated form, 5–44
  conditions controlling the use, 5–44
  function, 5–44
  general format, 5–44
  truncation of characters, 5–44
  when it is omitted, 5–44

K

key buffer
  allocating, 8–11
KEY clause
  SEARCH statement, 6–130
KEY data-names
  SORT statement
    entries containing OCCURS
      clause, 6–141
    rules governing, 6–140
key declaration
  purpose, 8–9
key declaration limit
  indexed files, 8–10
key file names
  explanation of duplicate, 7–12
key files
  allocating input/output areas, 4–17
  assignment to hardware device, 4–17
  blocking factor, 8–11
  high data area, D–2
  naming conventions, 8–9
  reducing I/Os, 8–11
key index
  purpose, 8–7
KEY IS phrase
  use in the OCCURS clause, 5–48
KEY items
  record sorting
    maximum specified, 6–140
key length
  in SORT
    total length exceeds 99
      bytes, 6–140
key of reference
  establishing for indexed files, 6–145
KEY phrase
  READ statement, 6–116
  START statement, 6–144, 6–145
KEY phrases
  in MERGE statement
    order of significance, 6–91
key size limit
  indexed files, 8–10
key value
  key file I/Os, 8–11

keys
  declaring, 8–11
keywords
  definition, 2–7
  example, 2–7
  format, 2–7
  Procedure Division
    placing, 1–11
  syntax
    definition, 1–3
  types, 2–7
KIND file attribute
  and SYNCHRONIZED attribute, 7–19
  explanations of TAPE options, 7–14
  overview, 7–14
  SCSI, 7–14

L

L option
  COMPILE statement, 9–2
  effect on the size of compiler work file, 9–24
label
  magnetic tape
    format, 5–26
label equation
  directing files to hardware devices, 9–6
  example, 9–6
  restriction on library files, 9–7
label equation statement
  discussion, 9–6
label records
  processing, OPEN statement, 6–102
LABEL RECORDS clause
  format, 5–25
  function, 5–25
  OMITTED option, 5–25
  STANDARD option, 5–25
label specifications
  Unisys
    using STANDARD in LABEL RECORDS clause, 5–25
LABELKIND file attribute
  function, 7–14
  MSSTANDARD, 7–14
  MSUSASII, 7–14
labels
  accessing data, 6–52
  assigning to areas of files, 7–1
  corruption of data, 7–15
  describing tape label, 7–14
  equating files to labels, 6–102
  in multi-programming environment, 9–6
  LABEL RECORDS clause and the presence, 5–25
  tape file
    manipulating with USE statement, 6–166
  tape
    accessing data, 6–52
language code table
  BCL, C–8
language elements
  discussion, 2–1
  required characters, 2–1
language structures
  character string manipulation, 2–2
  using separators, 2–2
LASTRECORD file attribute
  function, 7–15
  obtaining accurate results, 7–15
LASTSUBFILE attribute
  function, 7–26
LEADING phrase
  INSPECT statement
    replacing, 6–88
    tallying, 6–87
length mismatch
  parameters in calling programs, 9–16
LEV2IPC option
  calling called programs, 9–15
  discussion of run-time errors, 9–16
  DMS considerations, 9–15
  function and conditions, 9–15
  matching length of parameters, 9–16
  performance considerations, 9–15
level 1 IPC
  performance considerations, 9–15
level 2 IPC
  performance considerations, 9–15
  unpredictable results, 9–16
level indicator
  FD
    function, 5–19
  in file descriptions, 5–19
  SD
    function, 5–19
    relation to the file-name, 5–19
  using in qualification, 5–7

level number
function, 2–5
level numbers
assigning, 5–3
data description entry, format 1, 5–35
data descriptions
in Working-Storage, 5–45
Data Division
organizing data, 5–2
placement, 1–11
description of how level numbers are assigned, 5–35
discussion, 1–4
function, 5–45
general format, 5–45
memory area and many level-01 entries, 5–36
numbers 66, 77, 88, 5–36, 5–45
placement in area A and area B, 1–12
range that can be assigned, 5–35
ranging from 1 through 9
format, 5–45
special, 5–3
special, no true concept of level, 5–45
subordinate to FD or SD entries, 5–45
where a group item ends, 5–36
levels
referring to data in records, 5–2
subordinate
restrictions on VALUE clause, 5–70
levels of hierarchy
in qualifications, 5–8
LIB option
COMPILE statement, 9–3
LIBR option
function, 9–16
library files
contents, 6–72, 9–7
creating directly from CANDE EDITOR files, 6–71
creating with card input, 6–71
creating with COPY, 6–71
LIBR option, 9–16
LIBRFAMILY option, 9–16
ON clause of COPY statement, 6–67
overview, 9–7
required delimiters, 6–71
requirements for text-names, 6–72
restrictions, 9–7
single, containing multiple texts, 6–72
using, 6–72
library files (*See also* COPY statement), 6–71
library text
affect of REPLACING phrase of COPY, 6–68
determining syntactic accuracy, 6–67
determining text replacement, 6–69
format and creation, 6–67
processing COPY statement, 6–67
library-name
logical equivalence to literals in COPY, 6–67
LIBRFAMILY option
function, 9–16
function with COPY statement, 6–68
LIFO mechanism
benefits, in storage and retrieval functions, 6–135
LIMIT DMS USERS record
purpose, 8–13
limitations
physical
of file media, 8–1
LINAGE clause
definition of the footing area, 5–28
general format, 5–27
possible effect on printer backup files, 5–29
practical considerations, 5–29
required data-name references, 5–27
required values of elements within the, 5–27
size of logical versus physical page, 5–29
specifying depth of logical page, 5–27
value of elements within the, 5–28
WRITE statement, 6–174
linage functions
function of COIxxx intrinsic, 9–8
LINAGE-COUNTER
meaning of the value, 5–28
rules governing use, 5–28
when an OPEN statement is executed, 5–29
line size
determining with DISPLAY, 6–75
Linkage Section
allocation of space, 5–73
definition, 5–1
function in Data Division, 5–73
references to items, 5–73
structure, 5–73

list
  determining size, inquiry function, 6–135
LIST option
  effect of resetting, 9–17
  function, 9–16, 9–17
  inclusion of SUMMARY option, 9–20
list options
  & option control record, F–3
LISTDELETED option
  effect on compile listing, 9–12
  function, 9–16
lister construct
  Conversion Aid changes, F–46
LISTNEW
  & option control record, F–3
LISTOLD
  & option control record, F–3
LISTOMITTED option
  function, 9–16
LISTP option
  function, 9–17
LISTUN
  & option control record, F–3
literal phrases
  alphabet-name clause
    restrictions, 4–8
  syntax requirements, 4–8
literal subscriptions
  calculating, 5–15
literals
  alphabet-name clause, 4–8
  appearing in arithmetic expressions, 6–12
  CALL statement, 6–36
  Conversion Aid changes, F–30
  CURRENCY SIGN IS
    restricted characters, 4–9
  definition and construction, 2–11
  figurative constants in STRING statement, 6–148
  hexadecimal, 2–13
    when alphanumeric, 2–13
  in CANCEL statement, 6–58
  in INSPECT statement
    as figurative constants, 6–85
    requirements, 6–85
  in UNSTRING statement, 6–153
  in ZIP option of CALL SYSTEM, 6–56
  logical equivalent in COPY Statement, 6–67
  MULTIPLY statement, 6–97
  non-integer numeric
    illegal moves, 6–95
  non-numeric, 2–11
  numeric
    DECIMAL-BINARY option, 6–50
    discussion, 2–12
    in DISPLAY statement, 6–74
    maximum digits in arithmetic operations, 6–12
    use as subscripts, 5–11
  PERFORM statement, 6–104
  references to figurative constants, 2–11
  requirements in DIVIDE statement, 6–77
  requirements in STRING statement, 6–148
  STOP statement, 6–146
  using figurative constants, 2–10
  using in continued lines, 1–9
  VALUE clause
    substitution, 5–69
logical blocks
  BLOCK attribute, 7–9
logical conjunction
  AND (*See* AND)
logical correctness of file attributes
  responsibility for, 7–1
logical file
  associating with physical file, 7–12
  name of host processors, 7–25
  specifying maximum size of records, 7–15
logical files
  KIND attribute, 7–14
logical inclusive
  OR (*See* OR)
logical negation
  NOT (*See* NOT)
logical operators (*See* operators, logical)
logical page
  vertical positioning with WRITE statement, 6–170
logical program
  library files, 9–7
logical read
  FORCEIO attribute, 7–13
logical records (*See also* records, logical, 5–62
  availability and SAME RECORD AREA clause, 6–170

logical records (continued)
availability of
READ statement, 6–117
definition, 8–1
in RETURN statement
responsibility for size, 6–122
in Working-Storage, 8–1
relative files
using, 8–4
releasing, 6–170
SAME RECORD AREA clause, 4–31
size
within physical unit, 8–1
storing, space requirements, 6–172
lowercase words
notation
discussion, 1–3

M

magnetic tape
and completion of USE
procedure, 6–164
file organization, 8–2
format, 5–26
label format, 5–26
magnetic tape files
recording density, specifying, 7–11
MAIN dollar option
binding, 10–2
main module
binding
declaring, 10–2
MAIN option
function, 9–14, 9–17
MANxxx intrinsic
function, 9–8
MAP dollar option
debugging aid, D–1, D–2
in object code file
function of, D–2
program example, E–10
MAP option
function and conditions, 9–17
reference to example, 9–17
mapping
string of digits or characters, 6–51
margins
page, specifying top and bottom, 5–27
mass storage data record
accessing with SEEK, 6–131
mass storage device
logical records
storing, 6–172
mass storage files
access methods, 8–2
Conversion Aid changes, F–24
I-O phrases and OPEN
statement, 6–100
LABEL RECORDS clause, 5–25
labelling, 5–25
locating address of, 6–131
LOCK and LOCK ACCESS options of
OPEN, 6–101
logical replacement of existing
records, 6–124
making records available with
READ, 6–112
media, 8–2
random access
effect of WRITE, 6–171
sequential access
effect of WRITE, 6–171
size
within physical unit, 8–1
using ACTUAL KEY phrase, 4–19
WRITE operation
INVALID KEY condition, 6–171
mass storage media
file organization types, 8–2
MAXCENSUS and CENSUS attributes
effect on OUTPUTEVENT
attribute, 7–27
MAXCENSUS attribute
function, 7–26
MAXRECSIZE attribute
as a port attribute, 7–26
as a subport attribute, 7–26
MAXRECSIZE file attribute
function, 7–15
range of assigned values, 7–15
related attributes, 7–15
size relative to BLOCKSIZE, 7–15
MAXSUBFILES attribute
default value, 7–26
function, 7–26
MCP
mix table, 6–43
passing control information to, 6–49
MCP EOF pointer
purpose, 8–10
relative file
after opening I/O, 8–5

MCS option
  syntax, 9–17
  types, 9–17
memory
  allocated by the MCP
    size of, 9–3
  increasing for compiler, 9–3
  SAME clause, 4–31
  saving using segmentation, 6–7
memory allocation
  determining, 4–3
memory area
  multiple level 01 entries and
      re-definition of the
      same, 5–36
memory area allocation
  using SAME SORT AREA, 4–32
  using SAME SORT-MERGE
      AREA, 4–32
memory requirements
  compiler work files, 9–24
  reducing, 9–24
  returned by the DC keyboard command
      and HIGH MEMORY, 5–42
MEMORY SIZE clause
  Environment Division
    effect on SORT file size, 6–141
  MODULES option, 4–3
  overview, 4–3
  WORDS option, 4–3
memory use
  programs with several overlay
      areas, 6–8
MERGE and ICM
  CLEAR dollar option, 9–11
merge file
  naming files, 4–14
MERGE operation
  obtaining records from, 6–122
MERGE option
  limitations, 9–18
  syntax, 9–18
MERGE statement
  character positions compared to
      records, 6–90
  data items and OCCURS clause, 6–91
  data items identified by KEY
      data-names, 6–91
  DESCENDING phrase, 6–91
  describing file-name, 6–90
  file-name entries, descriptions, 6–90
  file-name from multiple file reel, 6–91
  files not in open mode, 6–98
  format, 6–89
  function, 6–89
  KEY phrases
    ASCENDING phrase, 6–91
    DESCENDING phrase, 6–91
  operand comparisons, 6–92
  output procedure
    contents, 6–91
    description, 6–91
  process when output procedure is
      specified, 6–91
  representation of output
      procedure, 6–90
  restriction on appearance in input
      procedure, 6–142
  restriction on repetition of
      file-names, 6–90
  rules pertaining to data-names, 6–90
  specifying GIVING phrase, 6–91
  USING phrase
    maximum file-names
      specified, 6–90
merged records
  obtaining MERGE operation, 6–122
message header
  ISC control commands, 4–16
message text size
  function of MAXRECSIZE, 7–26
messages
  COFLTR, F–53
  determining size
    RECEIVE statement, 6–119
  from ODT during WAIT
      operations, 6–168
  input
    specifying number that can be
      queued, 7–26
  NO MEM FOR STOQ, 7–27
  queued from subport files, 7–24
  sizing message to be sent, 6–134
  subport
    storage, 7–27
  suppressing correction warning
      messages, 9–12
  within queues, SEND
      statement, 6–133
minimum memory size
  MEMORY SIZE clause, 4–3

mix
  multiprogramming
    function of RECEIVE
      statement, 6–118
  number of programs in, 6–45
mix number
  format, 6–45
  obtaining, 7–21
MIX option
  CALL INTRINSIC statement, 6–43
mix table
  contents, 6–43
  returning information to calling
    program, 6–43
MIXID option
  CALL INTRINSIC statement, 6–45
MIXNUM option
  CALL INTRINSIC statement, 6–45
MIXNUMBER task attribute
  explanation, 7–21
MIXTBL option
  CALL INTRINSIC statement, 6–43
MLOG
  next file number, 6–48
  next record pointer, 6–48
mnemonic attribute relation condition
  abbreviation, 7–2
  format, 6–17
  general format, 7–2
  placement, 7–2
mnemonic attribute values
  names, 7–2
mnemonic file attributes (*See also* file
    attribute, mnemonic), 7–2
  comparisons involving, 6–17
mnemonic name
  function, 2–5
mnemonic names
  Special-Names paragraph
    purpose, 4–2
mnemonic values
  for FILESTATE attribute, 7–25
mnemonic-attribute-identifier
  and mnemonic file attributes, 6–17
mnemonic-attribute-value
  and mnemonic file attributes, 6–17
mnemonic-names
  relating to ODT, 4–6
  switch clause, 4–6
  WRITE statement, 6–174
mode instructions
  tracing of all normal, 6–56
modular programs
  calling other modules, 10–2
  creating, 10–1
  dollar options, using, 10–3
  entry points, 10–1
  ICM dollar option, setting, 10–2
  MAIN dollar option, setting, 10–2
  main module
    declaring, 10–2
  module functions
    defining, 10–1
  passing information and
    parameters, 10–1
  shared data areas
    defining, 10–1
modules
  calling other modules, 10–2
  placement in a program, 6–53
  when two or more modules refer to the
    same data item, 5–38
monitoring
  transfers of control, D–2
MOVE ALL literal
  examples, 2–10
MOVE statement
  CORRESPONDING phrase, 6–94
  data conversions, 6–95
  data transfer, 5–20
  elementary moves, 6–94, 6–95
  elements, description, 6–93
  example, 6–93
  format 1, 6–93
  format 2, 6–93
  function, 6–93
  illegal elementary moves, 6–94
  in numeric function, 6–25
  intermediate data item, using, 6–93
  receiving item, 6–95
  relationship to ACCEPT
    statement, 6–31
  sending and receiving areas, 6–93
  sending items, 6–95
  table of valid types, 6–94
  truncation, 6–96
  using with hexadecimal literals, 2–13
moves
  elementary, 6–94
  implied, 6–114
  truncation, 6–96
MSSTANDARD
  LABELKIND file attribute, 7–14

MSUSASI
  LABELKIND file attribute, 7–14
multifile tape
  assigning multi-file-id, 7–12
  searching for a file, 7–20
multi-file-identification
  assigning with FAMILYNAME, 7–12
  guard file, 7–17
MULTIPLE FILE clause
  POSITION clause, 4–33
  using, 4–33
multiple file reel
  restriction on file-names in SORT, 6–140
multiplication operator (*), 6–13
MULTIPLY statement
  composite operands, 6–97
  format 1, 6–97
  format 2, 6–97
  function, 6–97
  list of common features, 6–28
  literals, 6–97
  size error condition, 6–26
multi-programming environment
  equating label names, 9–6
multiprogramming mix
  communication between programs, 6–132
  function of RECEIVE statement, 6–118
MUSTBACKUP option
  BACKUPPERMITTED attribute, 7–9
MYHOSTNAME attribute
  function, 7–26
MYNAME attribute
  function, 7–26
  using null attribute, 7–23
MYUSE file attribute
  function, 7–15

N

names
  considerations when qualifying, 5–8, 6–4
naming conventions
  key files, 8–9
native character code set
  definition, 4–8
native character set
  value of literal, 4–8
native collating sequence
  characters, 4–9
  identification with alphabet name, 4–8
  using, 4–4
NATIVE phrase
  alphabet name, 4–8
nested IF statement
  description, 6–82
NEW
  effect of CLEAR dollar option, 9–11
NEW option
  syntax, 9–18
New Reserved Words
  Conversion Aid changes, F–37
NEWID option
  function, 9–19
  limitations, 9–19
NEXT phrase
  of READ statement, 6–114
  READ statement, 6–112
NEXT SENTENCE option
  READ statement, 6–112
NEXT SENTENCE phrase
  with RETURN, 6–123
NIL pointer dereference error
  description of, B–1
NN
  COFLTR option, F–12
NO BACKUP option
  function, 4–16
NO BUFFER AVAILABLE condition
  function of MAXCENSUS attribute, 7–26
NO FILE message
  OPTIONAL attribute, 7–16
NO REWIND option
  access to records, 7–12
NOBOUNDS option
  at execution of SET, 6–136
  function, 9–19
  function in bounds checking, 5–13
  restrictions, 9–19
NOBUFFER message
  CENSUS has reached MAXCENSUS, 7–27
nondeclarative procedures
  references to in USE procedure, 6–164, 6–166
nonfatal
  error messages, B–2

Non-Mass Storage Files
  Conversion Aid changes, F–24
nonnumeric comparisons
  determining truth value, 4–4
nonnumeric literals
  alphabet-name clause, 4–8
non-numeric literals
  definition, 2–11
  examples, 2–11
  format requirements, 2–11
  quotation marks as delimiters, 2–2
  separators in, 2–3
  VALUE clause
    limits set by PICTURE, 5–70
  value of literal, 4–8
nonshared indexed files
  access routines, 8–12
  accessing, 8–9
  allocating buffers, 8–12, 8–16
  buffer recommendations for multiple keys, 8–17
  buffer recommendations for one key only, 8–17
  consequence of FORCEIO, 8–12
  declaring, 8–11
  declaring alternate keys, 8–13
  duplicate key values, 8–18
  failure to CLOSE, 8–18
  file loads and writes, 8–18
  implementation considerations, 8–11
  invoking FORCEIO, 8–12
  key buffer recommendations, 8–17
  opening, 8–12
  random inquiry of buffers, 8–17
  random update of buffers, 8–17
  random updates and buffer size, 8–18
  reading and writing modified block, 8–12
  recovery, 8–18
  resetting data or key block defaults, 8–12
  REWRITES with multiple keys, 8–18
  sequential update of buffers, 8–17
  shared and nonshared mode, 8–8
  specifying key buffers, 8–12, 8–16
NORESEQ
  & option control record
    remaining options, F–4
NORETURN option
  function, 9–19
NOT
  function in arithmetic expressions, 6–15
NOT operator
  explanation, 6–20
notation
  asterisk (*)
    purpose, 1–8
  brackets ([]) and braces ({}), 1–4
  commas and semicolons, 1–5
  continuation lines
    example, 1–9
  debugging lines
    example, 1–10
  ellipsis
    example, 1–5
  level numbers, 1–4
  lowercase words
    discussion, 1–3
  quotation mark as separator, 2–2
  syntax elements
    description, 1–3
  uppercase words
    discussion, 1–3
  using $ sign, 1–11
  using braces and brackets
    example, 1–4
  using ellipsis marks and [] or {}, 1–4
  using hyphen (-) in indicator area, 1–9
  using periods in code, 1–5
  using spaces in code, 1–5
  using stroke (/) in indicator area, 1–8
  words, 1–3
    example, 1–3
NOTE statement
  Conversion Aid changes, F–37
null attribute
  definition, 7–23
number mismatch
  parameters in calling programs, 9–16
numeric
  data description entries (*See* data description entry)
numeric data
  method of editing, 5–54
  to define an item as, 5–49
numeric data description entry
  and the operational sign, 5–64

numeric data item
range of values, 2–14
numeric data items, 5–6
defining alignment, 5–8
SUBTRACT statement,
function, 6–151
numeric file attributes (*See also* file
attributes, numeric), 7–2
"1 relative", 7–2
numeric functions
format, 6–25
OFFSET function, 6–26
using in
arithmetic expression, 6–25
arithmetic statement, 6–25
MOVE statement, 6–25
Procedure Division, 6–25
numeric integer data items
unsigned
in the LINAGE clause, 5–27
numeric literals
alphabet-name clause, 4–8
as arithmetic operands, 2–12
definition, 2–12
examples, 2–13
ordinal number of character, 4–8
rules for formation, 2–12
signed
PICTURE character string, 5–70
use as subscripts, 5–11
VALUE clause
values set by PICTURE, 5–70
vs. non-numeric literals, 2–11
numeric operands
summing two or more, 6–33
numeric-edited
to define an item as, 5–50
numeric-edited data items
defining alignment, 5–8
method of editing, 5–54
using BLANK WHEN ZERO, 5–37

O

object code
generating, 9–22
object code file
debugging aids, D–2
MAP dollar option, D–2
object program
continuation after STOP, 6–146
Data Division
function, 5–1
function, 1–2
occupation of main memory, 9–24
segmentation requirements, 6–5
segments, 6–7
suspending execution, 6–167
suspension of execution, 6–146
object program item
stack, D–2
object program memory layout
debugging aids, D–1
object time switch (SW6)
effect on compilation, D–3
effect on debugging lines, D–3
USE FOR DEBUGGING section, D–3
OBJECT-COMPUTER name
& option control record
remaining options, F–4
OBJECT-COMPUTER paragraph
Conversion Aid changes, F–9
effect on debugging lines, 1–10
format, 4–3
function, 4–2, 4–3
MEMORY SIZE clause, 4–3
PROGRAM COLLATING SEQUENCE
clause, 4–4
SEGMENT-LIMIT clause, 4–4
STACK SIZE clause, 4–4
using STACK SIZE, D–2
valid system names, 4–3
occurrence numbers
element in a table
SET statement execution, 6–136
serial SEARCH operations, 6–128
OCCURS Clause
Conversion Aid changes, F–21
OCCURS clause
defining a three-dimensional
table, 5–9
defining a two-dimensional table, 5–9
definition of the subject entry
formats 1 and 2, 5–47
describing identifiers in SEARCH
syntax, 6–127
determining the highest subscript
value, 5–11
format 1, 5–46
format 2
requirements of data description
entries, 5–47
using table areas, 5–47

OCCURS clause (continued)
function, 5–46
general format 2, 5–46
importance in indexing, 5–12
restriction on use in a data description entry, 5–61
use, 5–46
using in repetitive tasks, 5–9
using subscription or indexing for data-names in the, 5–46
when it cannot be specified in a data description, 5–47
ODT
displaying messages on, 6–74
remote
displaying messages on, 6–74
ODT clause
Special-Names paragraph, 4–6
syntax example, 4–6
ODT-INPUT-PRESENT
effect on program suspension, 6–168
OF
loading an input file and OPEN, 6–101
using for qualification, 6–3
using in qualification, 5–7
OFFER phrase
of the OPEN statement
use with port files, 6–100
OPEN statement, 6–100
OFFSET function
as a numeric function, 6–25
data-names parameter, 6–26
OMIT option
function, 9–19
omitted images
listing in output, 9–16
OMITTED option
function in LABEL RECORDS clause, 5–25
ON clause
using in COPY statement, 6–67
ON ERROR clause
options, 6–140
ON ERROR option
in USE statement, 6–164
using with MERGE, 6–90
ON EXCEPTION clause
program suspension
with RECEIVE FROM TOP or RECEIVE FROM BOTTOM, 6–135
RECEIVE statement, 6–119
SEND statement, 6–134
ON EXCEPTION condition
RECEIVE statement, 6–120
SEND statement, 6–134
SEND statement (format 1), 6–133
ON OVERFLOW phrase
function, 6–38
specifying, 6–46
STRING data transfers, 6–150
UNSTRING statement, 6–156
ON pack-name
in displays from DISPLAY, 6–75
ON SIZE ERROR phrase
using in DIVIDE statement, 6–78
using OPTIMIZE option, 9–19
OPEN DEMAND statement
purpose, 6–101
OPEN EXTEND statement
function, 6–100
requirements, 6–100
OPEN file attribute
function, 7–16
OPEN FLOW statement
purpose, 6–101
OPEN operations
effect on SUBFILEERROR attribute, 7–27
OPEN statement
and execution of debugging section, 6–161
availability of file, 6–98
checking label records, 6–102
Conversion Aid changes, F–30, F–43
current record pointer, setting, 6–98, 6–99
DEMAND phrase, 6–101
effect on the setting of the LINAGE-COUNTER, 5–29
elements, requirements, 6–100
execution of the CLOSE statement, 6–62
EXTEND, 6–100
label records present, 6–102
format, 6–98
function, 6–98
I-O phrases for mass storage files, 6–100
label records not specified, 6–102
LOCK and LOCK ACCESS options, 6–101
mass storage files, 6–101

OPEN statement (continued)
OFFER phrase, 6–100
optional files, 6–101
order of execution and label processing, 6–102
OUTPUT phrase, 6–99
permissible I/O statements, 6–99
processing label records, 6–102
program suspension, 6–101
record area of file, effect on, 6–99
relationship to CLOSE statement, 6–99
REVERSED and NO REWIND phrases, 6–102
rewinding a file, 6–102
using OF and loading input files, 6–101
VAILABLE phrase, 6–100
writing labels, 6–102, 6–101
OPEN WITH LOCK
events, sequence, 6–101
purpose, 6–101
OPEN WITH LOCK ACCESS statement
purpose, 6–101
operands
comparison, 6–15, 6–16, 6–17
composite
limit on digits in MULTIPLY, 6–97
data transfers with DISPLAY
size of the sending item, 6–74
debugging section, 6–159
defining in USING phrase, 6–37
determining if numeric or alphabetic, 6–18
DISPLAY statement, 6–74
format considerations, 6–28
implied
CHARACTERS phrase of INSPECT, 6–86
in comparison operations, 6–86
in OCCURS clause, 5–14
in USING phrase, 6–37
lengths, 6–66
maximum size in arithmetic statements, 6–28
nonnumeric, 6–16
numeric, 6–16
summing two or more, 6–33
overlapping
discussion of process, 6–28
format considerations, 6–28
using moves, 6–28
sending
alphanumeric file attributes, 7–1
numeric file attributes, 7–2
SET statement
validity of combinations, 6–137
TALLYING and REPLACING, 6–86
unsigned numeric, 6–16
USING phrase, 6–2
operands in MERGE
comparing in relation condition, 6–92
operational sign
in a PICTURE clause without a SIGN clause, 5–64
presence in PICTURE clause, 5–51
presence of "S", 5–64
specified for COMPUTATIONAL data item, 5–52
specifying for DISPLAY data item, 5–51
using SIGN clause in Data Division, 5–64
operational signs
position and representation, 6–18
using ALPHABETIC class test, 6–18
using NUMERIC class test, 6–18
operators
logical
combining with conditions and using parentheses, 6–22
defined as logical connectors, 6–20
definitions, 6–20
table of combined conditions and, 6–22
truth table, 6–20
truth values, 6–20
using AND or OR in combined conditions, 6–21
valid combinations of NOT and OR, 6–22
OPTIMIZE option
function and conditions, 9–19
OPTIONAL file attribute
function, 7–16
OPTIONAL phrase
effect on file close, 6–62
effect on READ statement, 6–114
input files, 4–14
optional words
explanation, 2–8

options
  compiler
    discussion of $MAP and HIGH MEMORY, 5–42
  CRUNCH
    purpose, 6–71
  dollar
    compiling with $ICM set, 6–36
    $LEV21PC with CALL statement, 6–36
  ellipsis marks and repetition, 1–5
  MERGE statement, file close operations, 6–92
  NOBOUNDS
    SET statement, 6–136
  ON ERROR
    purpose, 6–90
  ON ERROR clause, 6–140
  $COPYBEGIN
    in COPY statement, 6–72
  $COPYBEGIN, in library programs, 6–71
  $COPYEND
    in COPY statement, 6–72
  $COPYEND, in library programs, 6–71
  $DMSICM
    restriction on use with HIGH MEMORY, 5–42
  $LEV21PC
    using with CANCEL statement, 6–58
  $OPTIMIZE
    effect on size error conditions, 6–27
OR operator
  explanation, 6–20
ORGANIZATION clause
  using, 8–4
ORGANIZATION clauses
  function, 4–19
ORGANIZATION IS SEQUENTIAL clause
  function, 4–19
OUT access to files
  explanation, 7–18
output
  program, 9–22
output file
  backup device for, 7–9
output files
  making sort records available in, 6–138
  random access mode in SEEK, 6–131
  symbolic, NEW option, 9–18
output listing
  causing immediate page break, 9–19
  inclusion of omitted items, 9–19
  overview, 9–22
  specifying lines per page, 9–19
output options
  & option control record, F–3
OUTPUT phrase
  OPEN statement, writing labels, 6–102
output procedure
  MERGE statement, description, 6–91
  representation in MERGE statement, 6–90
  restriction on using RETURN statement, 6–123
  SORT or MERGE
    use of RETURN, 6–122
  SORT statement
    restrictions on procedural statements within, 6–143
    restrictions on SORT and MERGE statements, 6–143
    RETURN statements in, 6–143
output source files
  blocking factors, 9–5
output symbolic file
  $NEW and DELETE, 9–12
OUTPUTEVENT attribute
  conditions for setting, 7–27
  equivalence to WRITE-OK, 7–27
  function, 7–27
overflow
  checking in intermediate data items, 6–66
  sort work file
    EOF NO LABEL message, 6–140
  truncation of index-name in SET execution, 6–136
  using ON SIZE ERROR clause, 6–66
overflow condition
  situations related to UNSTRING that cause, 6–155
overlapping operands (*See* operands, overlapping)
overlay
  requirements of object program, 6–5

overlay areas
how they are occupied, 6–8
multiple
example, 6–8
how memory is used, 6–8
multiple in memory, 6–7
passing control to segment not in memory, 6–8
rules, 6–8
segments in, 6–7
size, 6–7
transfer of control between, 6–7
overlayable segments
compromise between program size and execution speed, 6–7
fixed
in ALTER statement, 6–35
overlayed segments
debugging aids, D–2
overlaying segments
discussion of fixed permanent segments, 6–5
during execution, 6–5
fixed overlayable segments, 6–5

P

PACKIN
& option control record
input options, F–3
PACKOUT
& option control record
output options, F–3
page
determining line number where device is positioned, 5–28
format specifications
using LINAGE clause, 5–27
logical
default size if not specified, 5–27
definition of a, 5–27
positioning the file at the beginning, 5–29
specifying number of lines, 5–27
size of logical page versus size of physical page, 5–27
page break
on output listing, 9–19
PAGE option
function, 9–19
page overflow condition
automatic, 6–173
simultaneous occurrence of end-of-page, 6–173
PAGESIZE option
function and default value, 9–19
PAPERTAPE
using, 4–16
paragraph headers
description, 1–11
paragraph name
function, 2–5
paragraph names
equating, 2–5
paragraph syntax
example, 6–3
paragraph-names
duplicating, 6–4
location on coding form, 6–2
qualification, 6–2
qualification using section name, 6–4
paragraphs
definition, 6–2
duplicate names, 6–2
in procedures, 6–1
performing, 6–104
series, documenting logical end-point, 6–79
parameters
matching length of, 9–16
moving the value, 6–43
parity errors
using ON ERROR clause, 6–140
using ON ERROR option, 6–90
PARITY file attribute
function, 7–16
values, 7–16
Pascal
interlanguage binding
without stand-alone binder, 6–36
patch images
listing when LIST is reset, 9–17
PERFORM module
documenting logical end-point, 6–79
PERFORM statement
accessing elements of table, 6–111
AFTER option, 6–107
working with multi-dimensional table, 6–111
augmenting values, 6–107
BY phrase, 6–104

PERFORM statement (continued)
conditional expressions, 6–104
controlling times procedures are executed, 6–107
elements, description, 6–104
entries in program's stack area, 6–106
evaluation of conditions, 6–108
execution of a referenced procedure and the associated debugging section, 6–159
execution of debugging section, 6–163
format 1, 6–103
format 2, 6–103
format 3, 6–103
format 4, 6–104
giving value to index-name, 5–12
function, 6–103
how they operate, 6–106
implicit return mechanism, activating, 6–105
in input procedure for SORT, 6–142
index-names
in FROM phrase, 6–105
in VARYING or AFTER phrases, 6–105
procedure-names, relationships, 6–105
procedures referenced by, 6–106
return mechanism, 6–106
transfer of control, 6–105
UNTIL phrase, 6–107
using, 6–106
values of items named in VARYING or AFTER, 6–111
VARYING phrase
example of logic flow, 6–108
performance
attributes that affect, 7–28
effect of FORCEIO, 8–12
performance considerations
level 1 IPC, 9–15
level 2 IPC, 9–15
performance degradation
relative files
FORCEIO, 8–6
when using HIGH MEMORY programs, 5–43
performance issues
using NOBOUNDS dollar option, 5–13
performed procedures
situations that can lead to a program failure, 6–106
periods (*See also* notation and special characters), 1–5
Conversion Aid changes, F–41
phrases
CORRESPONDING
explanation, 6–27
frequently used, 6–26
resultant-identifier in, 6–26
ROUND
effect on SIZE ERROR, 6–26
ROUNDED
function, 6–26
SIZE ERROR
ADD with CORRESPONDING, 6–27
explanation, 6–26
specifying, 6–26
SUBTRACT with CORRESPONDING, 6–27
physical aspects
files, 8–1
physical file
indicating location on subsystem, 7–19
physical read
FORCEIO attribute, 7–13
physical records (*See* records, physical)
physical size
specifying character size, 7–12
physical unit
size of logical records, 8–1
PICTURE character string
avoiding truncation of data, 5–56
COMPUTATIONAL
allowed contents, 5–67
numeric character positions, 5–56
representing floating insertion editing, 5–55
representing zero suppression, 5–56
signed numeric literals, 5–70
suppression of leading zeros, 5–56
using floating insertion editing, 5–55
PICTURE character strings
combinations of symbols, 5–50
discussion, 2–14
UNSTRING statement, 6–154
PICTURE character strings (*See* character strings)

PICTURE clause
abbreviation, 5–49
character string
maximum number of characters, 5–49
character string symbols, 5–50
containing "S"
SIGN rules, 5–64
Conversion Aid changes, F–20
data size, 5–50
descriptions of the five data categories, 5–49
discussion of required specifications, 5–35
editing rules, 5–53
function, 5–49
general format, 5–49
in MOVE statements, 6–94
S and SIGN clause, 5–51
placement of *S*, 5–51
representing currency symbol, 4–9
rules controlling precedence, 5–58
symbols that can appear only once, 5–50
table of editing application, 5–58
use, 5–49
using / (slash) in character string, 5–53
using 0 (zero) in character string, 5–52
using A in a character string, 5–50
using asterisk (*) in character string, 5–53
using B in a character string, 5–50
using currency symbol ($) in character strings, 5–53
using editing sign control symbols in strings, 5–53
using *9* in character string, 5–52
using *I* in a character string, 5–51
using *P* in a character string, 5–51
using *S* in a character string, 5–51
using *V* in character string, 5–52
using *X* in character string, 5–52
using *Z* in character string, 5–52
using non-floating and floating insertion symbols, 5–58
using period (.) in character string, 5–53
using SIGN clause, 5–64
using with BLANK WHEN ZERO clause, 5–37
PICTURE clause, using comma (,) in a character string, 5–53
PICTURE strings
replacement, 6–70
required characters, 5–58
pointer
current record
accessing records in relative files with READ, 6–116
effect of DELETE statement, 6–73
indexed files and READ, 6–113
position in START comparisons, 6–145
READ with undefined pointer, 6–114
relative files and READ, 6–113
sequential files and READ, 6–113
setting, 6–99
setting with OPEN statement, 6–98
end-of-file
output files in SEEK, 6–131
POINTER phrase
STRING statement, 6–150
UNSTRING statement, 6–155
effect on data transfers, 6–156
with ON OVERFLOW phrase in STRING data transfers, 6–150
pointers
using subscripts in tables, 5–11
POP option
function, 9–20
port and subport attributes
BLOCKSTRUCTURE
function, 7–24
CENSUS, 7–24
CHANGEDSUBFILE
function, 7–24
CHANGEEVENT
function, 7–24
COMPRESSION, 7–24
COMPRESSIONPOSSIBLE
function, 7–24
CURRENTRECORD, 7–25
descriptions, 7–23
DIALOGPROTOCOLLEVEL, 7–25
explanation, 7–22
FILESTATE
function, 7–25
list of mnemonic values, 7–25
HOSTNAME, 7–25
INPUTEVENT, 7–25
INTNAME, 7–26

port and subport attributes (continued)
LASTSUBFILE, 7–26
list, 7–23
MAXCENSUS, 7–26
MAXRECSIZE, 7–26
MAXSUBFILES, 7–26
MYHOSTNAME, 7–26
MYNAME, 7–26
OUTPUTEVENT, 7–27
string attributes, 7–23
SUBFILEERROR, 7–27
table summarizing, 7–22
TITLE, 7–27
YOURNAME, 7–27
YOURUSERCODE, 7–28
port attributes
setting, 7–22
port file
maximum subports opened, 7–26
name
using INTNAME to declare and reference the file, 7–26
port files
accessing records with READ, 6–116
ASSIGN clause, 4–18
declaring, 8–24
designation, 8–24
File-Control paragraph
format, 4–13
OPEN AVAILABLE, 6–100
OPEN OFFER, 6–100
OPEN statement
options, 6–100
restrictions, 6–100
subfile for matching, 6–100
preventing suspension of files in READ operations, 6–117
purpose, 8–24
restrictions on declaratives, 6–1
sharing
restrictions, 5–23
types, 8–24
updating the ACTUAL KEY, WRITE operation, 6–176
using ACTUAL KEY clause, 4–22
using WITH NO WAIT in CLOSE, 6–62
versus storage queue mechanism, 8–24
WITH NO WAIT phrase in READ statement, 6–116
write beyond boundaries, result, 6–175
WRITE operation, purging data, 6–176
primary file
definition, 9–7
name as CARD, 9–5
primary source file
$MERGE and $DELETE option, 9–12
primary storage
determining amount, 7–21
primary symbolic file
and INSERT clause of COMPILE, 9–3
default medium, 9–5
PRIMARYSTORAGE task attribute
explanation, 7–21
prime record key
key reference for retrieval with READ, 6–116
purpose, 8–7
value in WRITE operation, 6–175
value when REWRITE is executed, 6–125
print files
function of FORMID, 7–13
printed page
vertical placement
using ADVANCING, 6–172
using END-OF-PAGE, 6–172
printer backup files (*See* files)
printer files
considerations when using the LINAGE clause, 5–29
PRIVATE option
SECURITYTYPE attribute, 7–17
PRNTIF
internal and external file names, F–2
PRNTOF
internal and external file names, F–2
problems
compiler, documenting, 1–2
Procedure Division
ADD statement, 6–33
CALL INTRINSIC statement, 6–39
CALL MODULE statement, 6–53
CALL statement, 6–36
CALL SYSTEM statement, 6–55
CHANGE statement, 6–59
changing labels
warning, 7–15
CLOSE statement, 6–61

Procedure Division (continued)
commas and semicolons
using, 1–5
common phrases, 6–26
complex conditions
order of evaluation, 6–24
COMPUTE statement, 6–65
conditional expressions, 6–15
condition-name condition, 6–19
Conversion Aid changes, F–23
Conversion Aid messages, F–56
COPY statement, 6–67
data references by calling program, 6–2
DECLARATIVES syntax, example, 6–3
DELETE statement, 6–73
description of arithmetic expressions, 6–12
DISPLAY statement, 6–74
DIVIDE statement, 6–76
error messages, B–8
example of correct syntax, 6–2
example of paragraph syntax, 6–3
examples of required formats
format 2, 6–3
EXIT statement, 6–79
file attribute identifiers, 7–3
function, 1–2
function of the CANCEL statement, 6–58
GO statement, 6–81
header
restriction on data entries declared with HIGH MEMORY clause, 5–42
IF statement, 6–82
INSPECT statement, 6–83
keywords
placing, 1–11
MERGE statement, 6–89
MERGE statements in declaratives, 6–90
modification of file attributes, 7–3
MOVE statement, 6–93
MULTIPLY statement, 6–97
naming a paragraph, 2–5
naming a section, 2–6
OPEN statement, 6–98
overview, 6–1
PERFORM statement, 6–103
placing keywords, 1–11
qualification using IN or OF, 6–3
READ statement, 6–112
RECEIVE statement, 6–118
references to data items, 5–73
RELEASE statement, 6–121
required formats, 6–3
restriction on specifying data-names with USING, 6–3
restrictions on transfers of control, 6–91
restrictions on transfers of control and SORT operations, 6–142
RETURN statement, 6–122
REWRITE statement, 6–124
rules for statement formats, 6–28
SEARCH statement, 6–126
SEEK statement, 6–131
segmentation, 6–5
SEND statement, 6–132
sentences
types, 6–10
SET statement, 6–136
SORT statement, 6–138
specifying numeric functions, 6–25
START statement, 6–144
statements
types, 6–10
STOP statement, 6–146
STRING statement, 6–148
structure, 6–2
SUBTRACT statement, 6–151
transfers of control within, 6–81
UNSTRING statement, 6–153
USAGE clause, restrictions, 5–67
USE statement, 6–158
using overlay areas, 6–7
USING phrase with CALL statement, 6–2
WAIT statement, 6–167
when execution begins, 6–2
WRITE BCT functions, 6–176
WRITE statement, 6–170
procedure-names
assigning to a point in program, 6–80
definition, 6–1
within debugging sections, 6–159
procedures
controlling, 6–106
processing of files (*See* file processing)

processing of programs
  bringing overlayable segment into memory, 6–7
  compromise between program size and execution speed, 6–7
processor number
  passing to MCP, 6–48
processor operations
  overlap with I/O operations, SEEK, 6–131
processor time
  obtaining amount, 7–21
  SORT key length exceeds 99 bytes, 6–140
PROGINFO option
  CALL INTRINSIC statement, 6–44
PROGINFOR option
  CALL INTRINSIC, 6–44
program
  modifying characteristics, 7–21
PROGRAM COLLATING SEQUENCE clause
  function, 4–4
  native, 4–4
  specifying alphabet-name, 4–7
program dictionary file
  function of $TADS, 9–20
program error handling
  function of USE statement, 6–158
program execution
  OBJECT-COMPUTER paragraph, 4–3
  suspending, 6–146
program failure
  establishing protection level, 7–16
  performed procedures, 6–106
  SORT operations, 6–140
program logic
  using PERFORM statement, 6–104
program memory dump
  run-time errors, 9–23
program module
  calling other modules, 10–2
  entry points, 10–1
program output
  discussion, 9–22
  discussion of time-of-compile, 9–23
  object code discussion, 9–22
  output listing, 9–22
  run-time errors, 9–23
program segments
  description of types, 6–5
program size
  initial
    and the DC keyboard command, 5–42
program stacks
  reference to information on, 6–106
program suspension
  conditions surrounding, 6–101
  conditions that cause, list, 6–101
  RECEIVE FROM TOP or RECEIVE FROM BOTTOM storage constructs, 6–135
  RECEIVE statement, 6–120
  SEND TO TOP or SEND TO BOTTOM storage constructs, 6–135
  using FORMID attribute, 7–13
  using the FM keyboard command, 7–13
  when program remains suspended, 6–101
  with WAIT statement
    determining length, 6–167
  WRITE WITH NO WAIT phrase, 6–175
program termination
  restriction on STOP RUN, 6–147
  SORT operations
    use of END, 6–140
  use of INTRINSIC ABORT call and STOP RUN, 6–147
  using ABNORMALSAVE of PROTECTION, 7–16
  using END, 6–90
PROGRAM-FAULT procedure
  restrictions on, 6–165
program-id paragraph
  format, 3–2
  function, 3–2
program-name
  effect of COMPILE SYNTAX option, 9–3
  executing, 9–3
  function, 2–6
  in CALL MODULE statement, 3–2
  in program-id paragraph, 3–2
  in source program, 3–2
programs
  modular
    creating, 10–1
  number in mix, 6–45

PROTECTION file attribute
ABNORMALSAVE, 7–16
default value, 7–16
function, 7–16
pseudotext delimiters
discussion, 2–3
PUBLIC
for shared indexed file, 8–14
PUBLIC option
SECURITYTYPE attribute, 7–17
punch card
EBCDIC, C–1
punch file
backup device for, 7–9
punch files
function of FORMID, 7–13
using BINARY option of EXTMODE, 7–12
using NONSTANDARD option of EXTMODE, 7–12
punched card files
file organization, 8–2
punched paper tape
file organization, 8–2
punctuation
COBOL syntax
example, 1–5
punctuation (*See* special characters and notation)
PURGE
function, 6–90
PURGE option
ON ERROR clause, 6–140
PUTLABEL option
CALL INTRINSIC statement, 6–52

Q

qualification
hierarchy, 5–7
of paragraph name, 6–4
process, 5–7
requirements for data-names, 5–7
rules for qualifiers, 5–8
using IN or OF, 5–7, 6–3
qualifications
in RENAMES clause, 5–62
qualifier connectives
function, 2–8
qualifiers
definition, 5–7
paragraph name qualification, 6–4
restriction on subscripting data-names, 5–8
restrictions, 5–7
rules for qualification, 5–8
queues
adding data, with SEND statement, 6–133
elements
allowed length in retrieval functions, 6–135
in main storage
use SEND and RECEIVE, 6–133
inquiry function
ACCEPT MESSAGE COUNT construct, 6–135
purpose, 6–135
list
naming elements in, 6–135
number of items in queue at once, 6–134
placement of data, 6–135
retrieving data from, 6–135
searches for items and RECEIVE statement, 6–120
searches from the bottom, 6–120
storage
meaning of a zero response, 6–32
number of entries, 6–32
sub-queues within, 6–133
using SEND TO BOTTOM, 6–135
using SEND TO TOP, 6–135

R

ramdom access mode
indexed files, 8–7
declaring files as shared, 8–23
file handling, 8–2
indexed files, 4–20
making records available from mass storage file, 6–112
order of access
in relative files, 8–4
order of file access, 4–19
reading files, 6–112
RECORD KEY data item, 4–21
relative files, 4–20

random access mode (continued)
sequential files
when physical read is performed, 6–116
setting ACCESS MODE clause, 4–19
using ACCESS MODE IS RANDOM, 4–19
read
nonselective
data for READ statement, 6–117
physical
sequential file accessed randomly, 6–116
READ statement
ACTUAL KEY declared for port file, 6–117
and execution of debugging section, 6–161
appearance in the USE procedure, 6–165
as a conditional statement, 6–10
AT END condition, 6–114
actions occurring, 6–115
required steps following, 6–115
Conversion Aid changes, F–30, F–44
current record pointer, position, 6–114
dynamic access mode files, 6–112
dynamic access to records in indexed files, 6–116
end of reel or unit operations, 6–114
executing, 6–112
execution, description, 6–115
execution of
updating the value of the FILE STATUS data item, 6–113
files in sequential access mode, 6–112
following REVERSED phrase of OPEN, 6–102
format 1, 6–112
format 2, 6–112
INVALID KEY condition, 6–116
function, 6–112
indexed files
key reference, 6–115
indexed files and position of the current record pointer, 6–113
indexed files, use with, 6–116
indexed I/O files, 6–112
INTO phrase, 6–114
INVALID KEY condition, 6–116
logical records described by more than one record description, 6–113
NEXT phrase, 6–112
NEXT SENTENCE in AT END phrase, 6–112
nonselective reads, 6–117
physical read, sequential files, 6–116
port files
ACTUAL KEY not declared, 6–117
position of current record pointer, 6–116
reading DCP files, 6–117
record availability for sequential files, 6–112
record retrieval for relative or indexed files with dynamic access mode, 6–113
relative files and current record pointer, 6–113
relative files, use with, 6–116
sequential files and current record pointer, 6–113
status of associated file, 6–112
unsuccessful execution, 6–114
current record pointer, 6–115
updating ACTUAL KEY data item, 6–115
USE procedure not specifed, 6–112
use with port files, 6–116
use with sequential files, 6–115
using for indexed file, 8–7
READERSORTER
assigning file to, 4–16
reader-sorter
constructs
Conversion Aid changes, F–46
reader-sorter files
OPEN DEMAND, 6–101
OPEN FLOW, 6–101
READLOCK option
CALL INTRINSIC statement, 6–46
READLOCKONLY
contrast with SEEKLOCKONLY, 6–46
READLOCKONLY option
CALL INTRINSIC statement, 6–46
using ON OVERFLOW or USE, 6–46
when ON OVERFLOW phrase must be specified, 6–46
when USE statement must be specified, 6–46

READLOCKWAIT
contrast with READLOCK, 6–46
contrast with SEEKLOCKWAIT, 6–46
READLOCKWAIT option
CALL INTRINSIC statement, 6–46
using ON OVERFLOW or USE, 6–46
when requested block already locked, 6–46
READ-OK
equivalence to INPUTEVENT in WAIT, 7–25
function in WAIT statement, 6–168
reccords
logical
determining the size of records, 5–30
RECEIVE FROM BOTTOM statement
function, 6–135
RECEIVE FROM TOP statement
function, 6–135
RECEIVE statement
determining size of messages, 6–119
EXCEPTION condition, 6–119
format 1, 6–118
format 2, 6–118
function, 6–118
imperative statements in EXCEPTION condition, 6–119
names of items in queue, 6–120
ON EXCEPTION condition, 6–120
RECEIVE FROM BOTTOM is specified, 6–120
reference to SEND statement, 6–119
STOQ, required setting, 6–120
STOQUE parameter block, 6–119
storage queue function, 6–119
suspended program, 6–120
synchronous communication between programs, 6–118
syntax elements, 6–118
using MCP's core-to-core option, 6–118
using MCP's storage queue capabilities, 6–118
receiving area
after UNSTRING data transfer, 6–157
UNSTRING statement, 6–156
receiving items
MOVE statement, descriptions, 6–95
record access
after executing DELETE statement, 6–73
relative files
READ statement, 6–116
selecting type, 7–8
sequentially accessed indexed files, 6–116
record area
data availability when INTO is used with RETURN, 6–122
effect of CLOSE statement, 6–62
effect of DELETE statement, 6–73
RECORD CONTAINS clause
and ANSIFILE option, 9–11
effect on file record handling, 9–11
function in the Data Division, 5–30
general format, 5–30
maximum record sizes for various types of files, 5–31
notes that apply to its use, 5–30
required reference of data-name 1, 5–31
when logical records are variable in length, 5–30
record descriptions
definition, 5–33
file-names in MERGE statement, 6–91
function, 5–2
hierarchy within, 5–33
level number 01, 5–45
using level numbers, 1–12
record handling
effect of RECORD CONTAINS clause, 9–11
record header
generating, 9–11
record items
alignment, 5–36
shared common
specifying, 5–38
RECORD KEY clause
prime record key, 4–21
references to data items, 4–21
RECORD KEY data items
prime
logical removal of records from indexed files, 6–73
RECORD KEY IS clause
using, 8–7

record layout
  example, 5–5
  level number construction
    example, 5–5
record length
  effect of ANSIFILE option, 9–11
record locking
  shared indexed files, 8–15
record numbers
  last record in physical file, 7–15
  purpose, 8–4
record pointer
  current
    accessing records in relative files with READ, 6–116
    effect of DELETE statement, 6–73
    effect of WRITE, 6–172
    indexed files and READ, 6–113
    position in START comparisons, 6–145
    READ with undefined pointer, 6–114
    relative files and READ, 6–113
    sequential files and READ, 6–113
    setting with OPEN, 6–98
    updating in indexed files, 6–113
    updating in relative files in READ operations, 6–113
record replacement
  character positions and REWRITE statement, 6–124
  indexed files
    REWRITE statement, 6–125
record retrieval
  sequential
    START statement, 6–144
record size
  comparison of logical to actual sizes, 5–30
  maximum, 6–172
  of the SORT file, 6–141
  physical
    maximum and minimum, 5–20
    specifying CHARACTERS, 5–20
    specifying RECORDS, 5–20
recording density
  specifying, 7–11
record-name
  function, 2–6
record-names
  Linkage Section
    requirements, 5–73
records
  access in indexed files, 8–7
  accessibility
    indexed files with READ, 6–113
    relative files with READ, 6–113
  accessing, 6–116
  assignment of level numbers, 5–35
  data (*See* data records)
  declaring, 5–21
  definition, 8–1
  elementary items, 5–2
  finding specific, 6–45
  fixed-length
    BLOCKSIZE attribute, 7–9
  identification in indexed files, 8–7
  identifying logically removed records, 6–73
  indexed files
    logically removing, 6–73
  key data items in indexed files, 8–7
  key index
    retrieval order, 4–20
  length
    CURRENTRECORD attribute, 7–25
  length on input files, 9–7
  level numbers
    use and assignment, 5–2
  locking, 8–15
  logical
    definition, 8–1
    described by more than one record description, 6–113
    determining the length, 5–30
    function of level numbers, 5–45
    grouping into physical record, 5–20
    reading, 6–112
    releasing, 6–170
    size requirements and the SORT statement, 6–140
    structure, 5–2
    using BLOCK CONTAINS clause, 5–20
    using DEPENDING clause, 5–30
    writing RENAMES entries to, 5–62
  logical replacement, 6–124
  maximum that can be sorted, 6–140
  merged
    obtaining from MERGE operation, 6–122
  movement of
    READ with INTO phrase, 6–114

records (continued)
moving and releasing
and the SORT statement, 6–142
organization in sequential files, 8–3
output procedure of MERGE, 6–91
physical
definition, 8–1
using BLOCK CONTAINS clause, 5–20
prime record key, 8–7
reading, 6–112
relative files
logically removing, 6–73
releasing locked, 6–46
selective compression, 7–24
smaller than maximum record size, 6–122
sorted
implicit functions on, 6–143
making available for processing, 6–143
obtaining from sort, 6–122
sorting, 6–138
specifying character size, 7–12
specifying format, 7–10
subdivisions, 5–2
transferring
as method of sorting, 6–138
to initial phase of SORT operation, 6–121
transfers to record area, 6–122
variable length
(*See* variable length records)
declaring, 5–21
vs. character positions in MERGE statement, 6–90
writing additional records to end of file, 6–100
recovery
corrupted indexed file, 8–18
indexed files, 8–10
nonshared indexed files, 8–18
relative files, 8–6
shared indexed files, 8–18
recursion
references to specific identifiers in STRING statements, 6–148
references to syntax elements in UNSTRING, 6–154
REDEFINES Clause
Conversion Aid changes, F–22
format, 5–60
function, 5–60
list of exception conditions, 5–60
placement, 5–60
placement in a data description entry, 5–35
requirements of entries in the, 5–60
restrictions, 5–8
restrictions in data description entry, 5–61
redefinition
amount of storage allocated, 5–60
beginning and ending, 5–60
example of an illegally redefined array, 5–61
examples of legally redefined arrays, 5–61
multiple
discussion, 5–60
of an elementary COMPUTATIONAL data item
by DISPLAY, 5–60
storage area
implicit, 6–122
redefinition of a data item
using a filler digit, 5–66
reference format
coding form, 1–6
reference formats
Conversion Aid changes, F–41
relation condition
format, 6–15
indexed data items, 5–68
relational operators, 6–16
required reference to a variable, 6–16
subject of the condition, 6–16
relation conditions
alphanumeric file attributes, 7–1
combined
abbreviating, 6–23
mnemonic attribute syntax, 6–17
mnemonic file attributes, 7–2
sequence
valid forms of abbreviation, 6–23
relation tests
making, 6–17
relational operators
as reserved words, 2–10
functions in arithmetic expressions, 6–16
using specific characters, 6–16

relative and indexed file Key files
resident segment area, D–2
relative file data block
purpose, 8–5
relative file handling routines
BSRxxx compiler intrinsic, 9–8
RELxxx compiler intrinsic, 9–8
relative files
access mode, 4–20
access routines, 8–2
accessing records with READ, 6–116
assigning to disk or diskpack, 4–17
block control information
effecting block size, 8–5
block size
limitation, 8–5
blocks
organization, 8–5
buffers
allocation and size, 8–5
control block size
limitation, 8–5
control of RANDOM access, 4–20
declaring, 8–4
DYNAMIC access, 4–20
FORCEIO
consequences, 8–6
FORCEIO block defaults
resetting, 8–6
format, 8–4
function, 8–4
function of FORCEIO attribute, 7–13
identifying logically removed
records, 6–73
implementation considerations, 8–5
INVALID KEY condition
WRITE operation, 6–171
INVALID KEY phrase and
REWRITE, 6–125
listing number of blocks, 8–6
logical positioning for sequential
retrieval of records, 6–144
logically removing records, 6–73
modified block
reading and writing, 8–6
NEXT phrase of READ, 6–112
ORGANIZATION IS RELATIVE, 4–19
record availability, 6–113
recovery, 8–6
sequential access, 4–20
sharing, restrictions, 5–23
simultaneous use by update
program, 8–6
START comparisons, 6–145
status of I/O operations, 4–25
using SHARED attribute, 7–18
value of status key 2, 4–25
variable length records, 8–5
write beyond boundaries,
result, 6–174
relative indexing (*See* indexing)
relative intrinsic
FORCEIO capability, 8–6
variable length records
supporting, 8–5
RELATIVE KEY clause
specifications, 4–20
START statement, 6–145
updating RELATIVE KEY data item
with READ statement, 6–115
RELATIVE KEY data items
logical removal of records, 6–73
logical replacement of records,
REWRITE statement, 6–125
relative organization
overview, 8–4
relative READ
physical I/O, 8–6
RELEASE statement
format, 6–121
FROM phrase, 6–121
function, 6–121
logical record, availability, 6–121
moving data, 6–121
record-name, requirements, 6–121
releasing records to sort, 6–121
restrictions, 6–121
when control passes from input
procedure, 6–121
RELxxx intrinsic
function, 9–8
REMAINDER phrase
definition, 6–77
in DIVIDE statement, 6–77
RENAMES clause
Conversion Aid changes, F–22
data-names that can be
qualified, 5–62
example, 5–63
function, 5–62

RENAMES clause (continued)
general format, 5–62
placing areas, requirements, 5–63
restrictions, 5–62
writing entries to logical records, 5–62
RENAMES entries
and level number 66, 5–45
replacements (*See* REPLACING phrase)
replacing
INSPECT statement, rules, 6–87
replacing entries
with COPY (*See* COPY statement)
REPLACING option
EXAMINE format, F–27
REPLACING phrase
cautions on using area A, 6–70
checking for misspellings, 6–70
COPY statement
comparison operation, 6–69
examples, 6–69
entries in area A, 6–70
examples of elements, 6–68
function in COPY statement, 6–68
matches during comparison operations, 6–70
of COPY
character strings within pseudo text, 6–68
description of pseudo text, 6–68
replacement of PICTURE string, 6–70
replacements in area A, examples, 6–71
restriction on replacing commas, 6–70
RESEQ option
& option control records, F–4
RESERVE clause
in ASSIGN clause, 4–18
RESERVE nn AREAS clause
using, 8–16
reserved memory
debugging aids, D–1
location, D–1
reserved words
as user-defined words, A–1
definition, 2–7
list, A–1
options within braces, 1–4
placement in Identification Division, 3–1
referencing figurative constant value, 2–10
referencing figurative constants, 2–9
types, 2–7
Unisys
Conversion Aid changes, F–39
usage restrictions, 2–7
using special character words, 2–10
RESET dollar option
purpose, 9–9
using with Boolean options, 9–9
resident code block constants
resident segment area, D–2
resident segment area
contents of, D–2
FIB, D–2
file record areas, D–2
relative and indexed file Key files, D–2
resident code block constants, D–2
resident segment code, D–2
working-storage data items, D–2
resident segment code
resident segment area, D–2
resultant-identifiers
definition, 6–26
effect on truncation, 6–26
storing results of ADD, 6–34
values when the SIZE ERROR phrase is specified, 6–27
RESULT-DESCRIPTOR option
ARM requirement, 6–43
CALL INTRINSIC statement, 6–43
retrieval functions
length of a queue element, 6–135
SEND statement, 6–135
return mechanism
deactivating, permanently, 6–106
effect of GO statement, 6–106
RETURN statement
as a conditional statement, 6–10
AT END condition, 6–122
contents of data items beyond range of current data record, 6–122
format, 6–122
function, 6–122
KEY clause in SORT or MERGE, 6–122
NEXT SENTENCE phrase, 6–123
record areas
availability of data with INTO, 6–122
records smaller than maximum record size, 6–122

RETURN statement (continued)
restriction on using in output procedure, 6–123
restrictions on storage areas, 6–122
restrictions on using, 6–122
size of logical record, 6–122
variable length records, 6–122
REVERSE option
DIRECTION attribute, 7–11
rewinding
rules that apply to, 6–102
REWRITE statement
as a conditional statement, 6–10
character positions in record replacement, 6–124
creation of duplicated values, 6–116
current record pointer, 6–124
discussion of the SAME RECORD AREA clause, 6–124
FROM phrase, 6–124
function, 6–124
general format, 6–124
INVALID KEY condition, 6–125
INVALID KEY phrase, 6–125
relative files, 6–125
restrictions, 6–124
sequential access mode, 6–124
syntax elements, 6–124
REWRITES
effect of buffer size on multiple keys, 8–18
RF Block (*See* relative file data block)
RLOG
next file number, 6–48
next record pointer, 6–48
ROUND phrase
effect on SIZE ERROR phrase, 6–26
ROUNDED phrase
function, 6–26
MULTIPLY statement, 6–97
rounding
of integer positions, 6–26
round-robin testing sequence
specification, 6–168
routine-name
function, 2–6
row number specification
EBCDIC, USASCII, BCL, C–1
run log entry
when IX1 is not zero and COMPILE-initiated program terminates, 6–147
RUN option
ON ERROR clause, 6–140
RUN phrase
STOP statement, considerations, 6–146
RUN statement
effect on bad blocks, 6–90
run-time errors
fatal
AREASIZE is zero, 7–8
logical attribute invalid, 7–1

S

SAME clause
overview, 4–31
restrictions, 4–31
SAME RECORD AREA clause, 4–31
using more than one, 4–31
SAME RECORD AREA clause
file associated with
and the REWRITE statement, 6–124
function, 4–31
sort-merge files in RELEASE operations, 6–121
WRITE statement, 6–170
SAME RECORD AREA (*See also* SAME clause), 4–31
SAME SORT AREA clause
using, 4–31, 4–32
SAME SORT-MERGE AREA clause
using, 4–32
SAVE option
COMPILE statement, 9–3
SAVECP option
function and default value, 9–20
SAVEFACTOR file attribute
function, 7–17
SAVEREC option
function, 9–20
scheduling programs
using ZIP in CALL SYSTEM, 6–56
screen design facility (SDF)
function of SDM710 compiler intrinsic, 9–8
MCS option, 9–17
SCSI
KIND file Attribute, 7–14
SYNCHRONIZED file attribute, 7–19

SD level indicator
  function, 5–19
SDF forms
  restriction on data items that reference, 5–42
SDIs
  list of shared file functions, 8–23
SDM720 intrinsic
  function, 9–8
SEA operation code
  SEARCH ALL operations with WHEN, 6–130
SEARCH ALL operations
  INDEXED BY phrase, 6–130
  predictability, 6–130
  WHEN condition specified, 6–130
SEARCH ALL statement
  giving value to index-name, 5–12
SEARCH operation
  serial, description, 6–128
SEARCH statement
  as a conditional statement, 6–10
  AT END phrase, 6–128
  Conversion Aid changes, F–31
  flowchart of a search operation, 6–128
  format 1, 6–126
  format 2, 6–126
  function, 6–126
  how execution works, 6–128
  index-names, 6–130
  occurrence numbers in serial search operations, 6–128
  OCCURS and INDEXED BY clauses, 6–127
  passing control, 6–130
  requirements, 5–15
  SEARCH ALL operations, 6–130
  serial search operations, 6–128
  syntax elements, 6–127, 6–130
  USAGE IS INDEX, 6–128
  using index data items, 5–15
  VARYING phrase, 6–127
  with indexed data items, 5–68
secondary file
  definition, 9–7
  DELETE option, 9–12
SECTION
  using in section headers, 1–11
section classification
  segment-numbers, 6–6
section ends
  in Procedure Division, 6–2
Section headers
  Conversion changes, F–32
section-name
  equating, 2–6
  function, 2–6
  qualifier for paragraph name using, 6–3
  using as qualifier for paragraph-name, 5–7
sections
  frequently referenced classification in segment logic, 6–6
  infrequently referenced classification in segment logic, 6–6
  series, documenting logical end-point, 6–79
  with same segment-number, 6–7
SECURED access to files
  explanation, 7–18
SECURITYFAMILY file attribute
  effect on guard file, 7–17
  function, 7–17
SECURITYGUARD file attribute
  changing file-id, 7–17
  function, 7–17
SECURITYTYPE file attribute
  default value, 7–17
  function, 7–17
  possible values, 7–17
  SECURITYFAMILY attribute, 7–17
  values, 8–14
SECURITYUSE file attribute
  changing, 7–18
  function, 7–17
  PUBLIC file access, 7–17
  values, 7–18
SEEK operations
  overlapping, 6–45
SEEK statement
  access mode of OUTPUT files, 6–131
  Conversion Aid changes, F–32
  following unsuccessful READ, 6–115
  format, 6–131
  function, 6–131
  required specifications, 6–131
  sequential access files, 6–131
SEEKLOCK option
  CALL INTRINSIC statement, 6–45

SEEKLOCKONLY option
  CALL INTRINSIC statement, 6–46
  contrast with READLOCKONLY, 6–46
SEEKLOCKWAIT
  contrast with READLOCKWAIT, 6–46
segment
  definition, 6–7
segment code and constants
  debugging aids, D–2
segment dictionary
  length and location, D–1
segmentation
  facility, 6–5
  in Procedure Division, 6–2
  limitations, 6–5
  Procedure Division, 6–5
  requirement for sections, 6–5
segmentation and overlay
  and compiler size, 9–24
segmentation logic
  frequency of use, 6–6
  logic requirements, 6–6
  transferring control with CALL MODULE, 6–6
segmented program
  appearance of CALL statement, 6–38
SEGMENT-LIMIT clause
  creation of fixed overlayable segments, 6–5
  OBJECT-COMPUTER paragraph, 4–4
  omitting, 4–4
  varying the number of fixed permanent segments, 6–5
segment-numbers
  digit designating overlay area, 6–7
  digits determining fixed or independent segment, 6–7
  discussion, 4–4
  for sections in declaratives section, 6–7
  format, 6–7
  frequency of section reference, 6–6
  function, 2–6
  in SEGMENT-LIMIT
    limitations, 4–4
  omitted from section header, 6–7
  permanent, 4–4
  references by ALTER statements, 6–35
  required range of values for, 6–7
  sections with the same, 6–7
  using in section classification, 6–6
segments
  fixed, 6–5
  fixed overlayable
    in ALTER statement, 6–35
    in overlay areas, 6–7
    logical equivalence to fixed segments, 6–5
    logical treatment, 6–5
    overlaying, 6–5
    range of segment numbers, 6–5
    state when called by a program, 6–5
    using SEGMENT-LIMIT clause, 6–5
  fixed permanent
    function, 6–5
    range of segment numbers for, 6–5
    restrictions on overlaying, 6–5
    storage during execution, 6–5
    using SEGMENT-LIMIT clause, 6–5
    when they can be overlayed, 6–5
  fixed portion, 4–4
  frequently called
    leaving in main memory, 6–7
  independent, 6–5
    definition, 6–6
    in overlay areas, 6–7
    limitations on ALTER statement references, 6–6
    organized into overlay areas, 6–6
    specifying, 6–6
  infrequently called
    defining as overlayable, 6–7
  overlayable
    compromise between program size and execution speed, 6–7
    using more than one, 6–7
  permanent, 4–4
    logic requirements, 6–6
  transfer of control between, 6–8
  types, 6–5
  without segment-numbers, 6–7
SELECT clause
  File-Control entry, 4–13
  requirements, 4–13
semicolon
  function, 1–5
SEND statement
  ACCEPT MESSAGE COUNT queue inquiry construct, 6–135
  adding data to queue, 6–133
  core-to-core option, setting, 6–133
  EXCEPTION condition, 6–133

SEND statement (continued)
format 1, 6–132
format 2, 6–132
function, 6–132
MCP STOQ option, setting, 6–134
named queue does not exists, 6–134
number of items in queue, 6–134
ON EXCEPTION condition, 6–134
reference for storage queue function, 6–32
specifying size of message, 6–134
STOQUE parameter block fields, 6–133
STOQUE parameter block structure, example, 6–133
synchronous communication between active programs, 6–132
syntax elements, 6–132, 6–133
using MCP's storage queue option, 6–132
using STOQUE mechanism, 6–134
SEND TO BOTTOM statement
function, 6–134
SEND TO TOP statement
function, 6–134
sending field
and UNSTRING, 6–154
sending items
MOVE statement, description, 6–95
SENSITIVEDATA file attribute
function, 7–18
sentences
compiler-directing, 6–10
definition and format, 6–2
function of imperative, 6–10
Procedure Division types, 6–10
SEPARATE CHARACTER phrase
and numeric data description entries, 5–65
Separators
Conversion Aid changes, F–40
separators
definition, 2–2
quotation mark, 2–2
rules, 2–3
rules for formation, 2–2
space, 2–3
SEQ nnnnnn +nnnnnn
function and default value, 9–20
SEQCHECK option
function, 9–20
sequence area
function, 1–8
sequence numbers
exception conditions, F–51
merging input from two files, 9–18
sequence of operations
modifying with ALTER statement, 6–35
SEQUENCE option
function, 9–20
sequenced files
combining two or more, 6–89
sequential access mode
ACCESS MODE clause, 4–19
file handling, 8–2
indexed files, 8–7
I/O statements and REWRITE, 6–124
making logical records available, 6–112
order of access
in relative files, 8–4
order of file access, 4–19
READ statement, 6–112
relative files, 4–20
restriction on INVALID KEY phrase, 6–125
restriction on using INVALID KEY phrase DELETE, 6–73
setting ACCESS MODE clause, 4–19
using SEEK statement, 6–131
SEQUENTIAL clause
using, 8–3
sequential file handling routines
INDxxx compiler intrinsic, 9–8
sequential files
blocking, 5–21
accessing records with READ, 6–115
deblocking, 5–21
discussion of access, 4–19
FILE STATUS clause, 4–22
format of File-Control paragraph, 4–10
INVALID KEY condition
WRITE operation, 6–171
making records available with READ, 6–112
multiple buffers and BLOCK attribute, 7–9

sequential files (continued)
NEXT SENTENCE option of READ, 6–112
OPEN EXTEND statement, 6–100
OPEN NO REWIND, 6–102
OPEN REVERSED, 6–102
ORGANIZATION IS SEQUENTIAL, 4–19
organization of records, 8–3
overview, 8–3
physical read operations, 6–116
record availability, 6–112
SEEK statement execution following unsuccessful READ, 6–115
SHARED attribute, 4–19
using NEXT SENTENCE phrase with RETURN, 6–123
using SHARED attribute, 7–18
sequential I/O module
function, 8–3
sequential organization
overview, 8–3
specifying, 8–3
sequential relationships
between procedure-names in PERFORM, 6–105
serial number
definition, 7–18
SERIALNO file attribute
function, 7–18
interrogation, 7–18
SET dollar option
example, 9–10
purpose, 9–9
using with Boolean options, 9–9
SET NDX TO statement
using in table references, 5–15
SET statement
at execution
discussion of abnormal termination, 6–136
decrementing values with DOWN BY, 6–137
evaluation of subscripting or indexing, 6–137
format 1, 6–136
format 2, 6–136
function, 6–136
giving value to index-name, 5–12
incrementing values with UP BY, 6–137
INDEXED BY clause, 6–136
relationship to SEARCH and index-names, 6–127
restrictions on index-names, 6–136
syntax, 9–9
syntax elements and references, 6–136
using index data items, 5–15
validity of various combinations of operands, 6–137
with indexed data items, 5–68
SETSWITCH option
CALL INTRINSIC statement, 6–43
shared
allocating key buffers, 8–14
shared data items
declaring as COMMON, 10–2
shared disks
locking blocks, 6–46
SHARED file attribute
function, 7–18
restrictions, 7–18
with indexed files, 7–18
with sequential files, 7–18
shared file functions
accessing multiple programs or processors, 8–23
declaring and accessing shared files, 8–23
locking a block, 8–23
providing, 6–39
setting MCP options, 8–23
when to declare as shared, 8–23
shared files
access exceptions, 8–24
declaring, 8–23
declaring and accessing, 8–23
declaring as shared, 8–23
error handling, 6–47
exception-handling procedures, 8–24
handling errors in DECLARATIVES, 6–47
implementation considerations, 8–13
locking, 6–46
required access, explanation, 7–18
STALEMATE condition, 6–47
USE procedure, 8–24

shared indexed files
  accessing, 8–8, 8–9
  allocating key buffers, 8–14
  concurrent updates, 8–14
  declaring, 8–13
  declaring alternate keys, 8–13
  declaring keys and descriptions, 8–13
  effect on indexed files, 8–8
  file attributes, 8–14
  hardware requirement, 8–13
  implementation considerations, 8–13
  ISAMDB, 8–13
  ISHxxx, 8–13
  limiting users, 8–13
  locking records, 8–15
  memory requirement, 8–13
  record locking example, 8–15
  recovery procedures, 8–18
  requirements, 8–13
  resolving deadlocks, 8–15
  security specification, 8–14
  shared and nonshared mode, 8–8
  size of ISAM handler, 8–15
  software requirement, 8–13
shared ISAM files
  declaring keys and descriptions, 8–13
  limitation on open files, 8–8
shared ISAM handler
  allocating key buffers, 8–14
  concurrent updates, 8–15
  detecting deadlocks, 8–15
  hardware requirement, 8–13
  limiting users, 8–13
  memory requirement, 8–13
  memory size and growth, 8–15
  record locking example, 8–15
  recovery, 8–18
  restrictions, 8–10
  terminating deadlocks, 8–15
shared system
  accessing multiple programs or processors, 8–23
SHRD
  purpose, 8–23
SHRP
  purpose, 8–23
SIGN clause
  function, 5–64
  general format, 5–64
  if usage of the data item is COMPUTATIONAL, 5–65
  if usage of the data item is DISPLAY, 5–65
  when specified for a group item, 5–64
sign condition
  format, 6–20
  function, 6–20
  NOT and algebraic tests, 6–20
  truth tests, 6–20
SIGN IS SEPARATE clause
  using CODE-SET clause, 5–22
  when to use with the CODE-SET clause, 5–64
simple condition
  definition, 6–15
simple conditions
  negated
    format, 6–21
    using parentheses, 6–21
  negation, 6–21
  using logical operator NOT, 6–21
SINGLE UNIT option
  DISKPACK files, 4–16
SINGLEUNIT file attribute
  default value, 7–19
  function, 7–19
size error condition
  using OPTIMIZE option, 9–19
SIZE ERROR phrase
  ADD with CORRESPONDING, 6–27
  DIVIDE statement, 6–26
  effect of $OPTIMIZE option, 6–27
  effect on ROUND phrase, 6–26
  function, 6–26
  MULTIPLY statement, 6–26
  overflow and setting index-names in SET, 6–136
  SUBTRACT with CORRESPONDING, 6–27
  when it applies, 6–26
sizes and limits
  compiler, 9–24
SLOG
  next file number, 6–48
  next record pointer, 6–48
Small Computer Systems Interface (*See* SCSI)
sort file
  ensuring sufficient length, 6–140
  naming files, 4–14
  size
SORT: intrinsic
  initiating, 6–140

SORT memory area
use of MEMORY SIZE as a guideline for allocation, 6–141
SORT statement
allocation of character positions for corresponding records, 6–140
Conversion Aid changes, F–32
data-names following KEY
order of significance, 6–141
file closing, options, 6–143
format, 6–138
function, 6–138
GIVING phrase, 6–143
implied transfer of control to declaratives, 6–143
input procedures, requirements, 6–142
maximum elements, 6–140
maximum files specified in USING phrase, 6–142
maximum records to be sorted, 6–140
moving and releasing records, 6–142
obtaining records from sort, 6–122
passing control when an input procedure is specified, 6–141
processing and releasing logical records, 6–143
referencing files that are not open, 6–98
relation to RELEASE operations, 6–121
releasing records to sort, 6–121
restriction on appearance in DECLARATIVES, 6–1
restriction on input procedure, 6–142
restrictions, 6–138, 6–139
size requirements for logical records, 6–140
sorted sequence with ASCENDING, 6–141
sorted sequence with DESCENDING, 6–141
syntax elements, 6–140
transfers of control, 6–142
using COLLATING SEQUENCE clause, 6–141
using ON ERROR clause, 6–140
sort work file
when overflow occurs, 6–140
sorted records
implicit functions on, 6–143
obtaining from SORT operation, 6–122
SORT/MERGE
collating sequence
COIxxx compiler intrinsic, 9–8
sort-merge
file description entry, 5–19
sort-merge files
format of File-Control paragraph, 4–13
maximum block size, 5–20
maximum record sizes for various types of files, 5–31
reading next record, 6–122
specifying BLOCK CONTAINS clause, 5–20
using RELEASE to write records to, 6–121
source code
conversion source code summary listing, F–58
expansion
exception conditions, F–50
source file
primary
$MERGE and $DELETE option, 9–12
secondary
$DELETE option, 9–12
source images
effect of $DOUBLE option, 9–12
effect of LIST option, 9–16
resequencing, 9–20
source input files
primary
using OMIT option, 9–19
secondary
using OMIT option, 9–19
source language
input media, 9–7
source language program
compilation and control, 9–1
source program
blank lines
placement, 1–12
compiling
processing COPY statements, 6–67
definition, 1–2
determining syntactic accuracy, 6–67
divisions
list, 1–2

source program (continued)
formation of text, 2–2
ICM files
creating, 3–2
incorporating copied text, 6–72
input files, 9–7
keywords
format, 2–7
level number entries, 5–3
level of syntax to be applied, 9–13
method of describing, 1–6
program-name
function, 3–2
relation to object program, 1–2
rules for writing, 1–6
table of federal syntax levels, 9–14
source template
indexed files recovery, 8–19
source text
storing as a source library file, 6–71
SOURCE-COMPUTER paragraph
computer-name clause, 4–2
Conversion Aid changes, F–8
format, 4–2
function, 4–2
Input-Output Section
function, 4–10
WITH DEBUGGING MODE clause, 4–2
spaces
function in code, 1–5
special characters
COBOL syntax
requirements, 1–5
special forms bit (FIBSPF)
resetting, 7–13
special forms flag
resetting with FORMID, 7–13
special level numbers
overview, 5–3, 5–45
special registers
contents, 2–8
Conversion Aid changes, F–32
function, 2–8
list, 2–8
SPECIAL-NAMES paragraph
alphabet-name clause, 4–7
clauses
order, 4–5
condition names, 2–4
Conversion Aid changes, F–10
DECIMAL-POINT IS COMMA
clause, 5–53
format, 4–5
function, 4–2, 4–5
using CHANNEL clause, 4–6
using ODT clause, 4–6
using switch clause, 4–6
SPOMESSAGE option
allowed input messages, 6–47
CALL INTRINSIC statement, 6–47
GIVING phrase, 6–48
USING phrase elements, 6–48
SRTUTL
accessing nonshared indexed
files, 8–11
accessing relative files, 8–5
stack
association with Boolean options, 9–9
compiler, 9–2
debugging aids, D–2
minimum size, D–2
stack area
entries from PERFORM
statement, 6–106
stack size
specifying during compile, 9–4
STACK SIZE
use in Object-Computer
paragraph, D–2
STACK SIZE clause
function, 4–4
using, 9–22
stalemate
exception-handling procedures, 8–24
STALEMATE condition
list of required actions, 6–47
STANDARD option
LABEL RECORDS clause, 5–25
standards
FEDLEVEL option, 9–13
START statement
as a conditional statement, 6–10
comparison types in KEY
phrase, 6–145
following unsuccessful READ, 6–115
format, 6–144
function, 6–144
indexed files
key of reference, 6–145
INVALID KEY condition exists, 6–145
KEY phrase, 6–144
mode of an open file-name during
execution, 6–144

START statement (continued)
syntax elements, 6–144
unsuccessful execution, 6–145
updating FILE STATUS data item, 6–145
when the INVALID KEY phrase must be specified, 6–144
statements
compiler-directing, 6–10
conditional, 6–10
examples, 6–10
IF statement, 6–82
definition and format, 6–2
elements
list, 1–3
function of imperative, 6–10
order of execution, 6–2
Procedure Division
types, 6–10
rules for format, 6–28
syntax
definition, 1–3
status digits
program error notification, 8–20
table of values, 8–20
STOP literal
STOP statement, considerations, 6–146
STOP RUN statement
and CLOSE statement, 6–62
clearing IX1, 6–147
restrictions, 6–146
STOP statement
considerations if RUN phrase used, 6–146
continuation of object program, 6–146
Conversion Aid changes, F–34
format, 6–146
function, 6–146
STOP literal specified, 6–146
STOQ option
MCP
requirement with RECEIVE, 6–120
purpose, 6–134
STOQ-INPUT clause
WAIT statement, 6–168
STOQUE MCP module
function, 6–134
types of calls, 6–135
STOQUE parameter block
entry-data-length-field, 6–32
example of structure, 6–32
example of structure in SEND, 6–133
name, 6–32
RECEIVE statement, 6–119
requests made of STOQUE, 6–135
SEND TO BOTTOM, function, 6–134
SEND TO TOP, function, 6–134
sending messages, 6–133
structure, example, 6–119
storage and retrieval functions
SEND statement, 6–135
storage area
redefinition, 5–60
storage areas
allocating
sort or merge files, 4–32
restrictions
INTO phrase of RETURN, 6–122
restrictions, using READ with INTO, 6–114
shared
logical records of file, 6–113
using overlapping operands, 6–28
storage medium
associating a file, 4–14
storage queue
capabilities
using RECEIVE statement, 6–118
explanation, 6–134
meaning of a zero response, 6–32
multiple transactions in, 6–134
number of entries, 6–32
storage queue functions
reference to SEND statement, 6–32
storage queue option
MCP
further references to, 6–135
required setting,SEND statement, 6–134
using SEND statement, 6–132
STRING statement
as a conditional statement, 6–10
data items whose usage is DISPLAY, 6–149
DELIMITED phrase with SIZE phrase, 6–149
DELIMITED phrase without SIZE phrase, 6–149
FOR phrase, 6–150
format, 6–148
function, 6–148

STRING statement (continued)
number of characters to be moved, 6–149
ON OVERFLOW phrase, 6–150
POINTER phrase, 6–150
receiving items, 6–148
restriction on ALL, 6–148
rules controlling transfers of data, 6–149
program example, E–1
sending items, 6–148
syntax elements, 6–148
use of literals in, 6–148
using SIZE phrase, 6–149
versus UNSTRING statement, 6–153
subfile index
updating ACTUAL KEY in READ, 6–117
SUBFILEERROR attribute
values, 7–27
subfiles
determining which subfile to read and READ with port files, 6–117
indicating which subfile index was read and the ACTUAL KEY, 6–117
maximum number for a port, 4–18
opening, with AVAILABLE phrase of OPEN, 6–100
subject of condition
definition, 6–16
subordinate levels
with VALUE clause, 5–70
subport
matching process, 7–26
YOURUSERCODE attribute, 7–28
subport and port attributes (*See* port and subport attributes)
subport dialogue feature
using COMPRESSION attribute, 7–24
subport file attributes
CENSUS attribute, 7–24
modifying, 6–59
setting, 7–22
subport files
compression of data sent between, 7–24
number of messages queued from, 7–24
specifying in WAIT statement, 6–168
using CURRENTRECORD attribute, 7–25
subport index
LASTSUBFILE attribute, 7–26
using CHANGEDSUBFILE attribute, 7–24
subport messages
storage, 7–27
subports
maximum that can be opened, 7–26
subport-to-subport protocol
determining version, 7–25
subroutines
executing
and program logic, 6–104
using PERFORM statement, 6–104
subscript
at object time, 5–14
definition, 5–11
example, 5–14
locating items in tables, 5–14
validation, 5–13
subscripted data-name
description, 5–11
subscripting
evaluation, in MOVE statement, 6–93
evaluation of UNSTRING statement, 6–154
example, 5–14
order of occurrence
in RETURN and MOVE operations, 6–122
versus indexing, 5–14
subscription
qualifying indexed data-name, 5–8
when data-names are used as qualifiers, 5–8
subscriptions
literal
calculating, 5–15
subscripts
format, 5–12
highest permissible value, 5–11
identifying data items in tables, 5–9
lowest possible value, 5–11
program termination, 5–11
requirements for value, 5–13
restrictions, 5–12
use in DEBUG-NAME, 6–162
when permitted, 5–13
SUBSYSTEM file attribute
DEFAULT option, 7–19
function, 7–19

SUBTRACT statement
effect on data items, 6–152
exceptions to required references, 6–152
format 1, 6–151
format 2, 6–151
format 3, 6–151
function, 6–151
list of common features, 6–28
subtraction operator (−), 6–13
SUMMARY option
function and conditions, 9–20
suspension of program (*See* program suspension)
SW6
debugging
purpose, D–3
definition, 4–2, 4–7
effect on compilation, D–3
effect on debugging lines, D–3
requirements WITH DEBUGGING MODE, 4–7
SWITCH
& option control record, F–5
Conversion Aid changes, F–34
switch clause
example, 4–7
Special-Names paragraph, 4–6
using mnemonic-names, 4–6
switch names
list, 4–7
switches
CALL INTRINSIC statement, 6–43
checking status, 4–6
determining the status, 6–19
implementer-defined, 2–5
list of values, 4–7
setting at program initiation, 4–7
switch-status condition
format, 6–19
function, 6–19
requirements for implementor-name, 6–19
when the result is true, 6–19
symbolic file
output
$NEW and DELETE, 9–12
symbolic input files
use of the MERGE option, 9–18
symbolic output files
creating a new, 9–18
inclusion of discarded items, 9–21
updated master, 9–22
writing dollar cards, 9–19
SYNCHRONIZED clause, 5–66
abbreviation, 5–66
function in Data Division, 5–66
general format, 5–66
restriction on use, 5–66
SYNCHRONIZED file attribute
explanation, 7–19
SCSI, 7–19
SYNCHRONIZED item
when referenced in the source program, 5–66
synchronous communication
between programs in same mix, 6–118
SYNTAX
& option control record, F–4
syntax
element descriptions, 1–3
syntax errors
effect on object code, 9–22
entry in run log when COMPILE-initiated program terminates, 6–147
message list, B–12
syntax levels
FEDLEVEL options, 9–13
table of federal, 9–14
SYNTAX option
COMPILE statement, 9–3
syntax rules
alphabet-name clause, 4–8
system dependent intrinisics (*See* intrinsics)
System Independent Intrinsics
description, 6–52
system library
using SAVE option of COMPILE, 9–3
system performance
attributes that affect, 7–28
system utilities
invocation by COBOL user program, 6–55
system-names
character set, 2–6
definition and function, 2–6
types, 2–6

T

table
  BCL language codes, C–8
  interchange codes, EBCDIC, C–4
  USASCII code set, C–2
table indexing
  effect on IX1, 6–147
table operations
  establishing reference points for, 6–136
table references
  indexing, 5–12
  subscripting, 5–11
  using an index-name, 5–12
  using SET NDX TO data-name, 5–15
table searching
  effect on IX1, 6–147
tables
  accessing elements, 5–10
  accessing elements with AFTER option of PERFORM, 6–111
  defining
    three-dimensional, 5–9
    two-dimensional, 5–9
  dimensions of
    associated with index-names, 6–127
  editing methods, 5–54
  function of subscripts, 5–9
  in Working-Storage, 5–10
  initialization, example, 5–11
  maximum dimensions, 5–10
  multidimensional
    AFTER option in PERFORM, 6–111
    SEARCH statement, 6–127
  OCCURS clause, 5–9
  one-dimensional
    defining, 5–9
    example, 5–9
  redefining, 5–11
  redefining elements, 5–61
  returning MPC mix table information, 6–43
  SEARCH operation
    identification of unique elements, 6–130
  searching, 6–126
  status digit values, 8–20
  subscripting, 5–11
  three-dimensional
    defining, 5–9
    example, 5–10
  two-dimensional
    defining, 5–9
  using in Data Division, 5–9
  using OCCURS clause when defining, 5–46
TADS option
  syntax, 9–20
tallying
  INSPECT statement
    rules, 6–87
TALLYING option
  EXAMINE format, F–26
  INSPECT statement, effect, 6–85
  INSPECT statement (format 3), 6–85
TALLYING phrase
  UNSTRING statement, 6–155
tape
  format, 5–26
  label format, 5–26
  multifile
    density, 7–11
  multireel
    density, 7–11
  primary work medium for a SORT file, 4–17
  variable-length records
    with SORT, 6–140
TAPE assignment
  absence of, 4–16
  specification of 7 CHANNEL, 4–16
tape files
  CREATIONDATE attribute, 7–10
  DENSITY value for, 7–11
  expiration date, 7–17
  interrogating SERIALNO, 7–18
  reading in reverse, 7–11
  recording density, specifying, 7–11
  TODAY-DATE and creation date, 7–10
  using PARITY and its values, 7–16
  using PARITY with EVEN, 7–16
tape label handling
  program example, E–16
tape labels
  accessing data, 6–52
  accessing with program, 6–52
  passing information, example, E–16
tape reader
  PAPERREADER option of KIND, 7–14

TAPEIN
& option control record
input options, F–3
TAPEOUT
& option control record
output options, F–3
tapes
assignment
7-track, 4–16
9 channel GCR, 4–16
9 channel phase-encoded, 4–16
9-track, 4–16
task
modifying characteristics, 7–21
task attributes
ACCUMIOTIME, 7–21
ACCUMPROCTIME, 7–21
explanation, 7–20
format, 7–21
function of COIxxx intrinsic, 9–8
MIXNUMBER, 7–21
PRIMARYSTORAGE, 7–21
TEMPORARY option
PROTECTION attribute, 7–16
terminal
time-sharing, displaying
messages, 6–74
termination
abnormal
and execution of SET, 6–136
invalid subscipt or index, 5–14
run-time errors, 9–23
testing sequences
specifying with USING, 6–168
text
incorporating in source
program, 6–67
text-name
function, 2–6
in COPY
limitation on length, 6–67
in library files, requirements, 6–72
THEN separator
Conversion Aid changes, F–40
THROUGH
vs. THRU, 4–8
THROUGH phrase
collating sequence, 4–9
THRU
vs. THROUGH, 4–8
time
decimal approximation, 6–31
TIME field
ACCEPT statement, 6–31
TIME OF RESPONSE
GETMCP option of CALL
INTRINSIC, 6–48
TIME register
Conversion Aid changes, F–33
TIME-NOW option
CALL INTRINSIC statement, 6–43
time-of-compile
discussion, 9–23
TIMER field
in ACCEPT statement, 6–31
TITLE attribute
default value, 7–27
definition, 7–27
TITLE option
function and default value, 9–20
title storage
using NEWID option, 9–19
TO
using in ADD statement, 6–33
TODAY-DATE
tape file creation, 7–10
TODAYS-DATE field
ACCEPT statement, 6–32
TODAYS-DATE register
Conversion Aid changes, F–33
TODAYS-NAME field
ACCEPT statement, 6–32
TRACE statement
Conversion Aid changes, F–35
TRACE-OFF option
CALL SYSTEM statement, 6–56
function, 6–56
TRACE-ON option
CALL SYSTEM statement, 6–56
transactions
multiple
in storage queue at same
time, 6–134
transfers of characters
in STRING, movement, 6–150
transfers of control
AT END condition recognized, 6–115
explicit
to outside output procedure of
SORT, 6–143
explicitly and implicitly, using
PERFORM, 6–103
implicit, 6–105

transfers of control (continued)
implied
in SORT operations, 6–142
output procedure, MERGE statement, 6–91
PERFORM statement, 6–105, 6–106
source program monitoring, D–2
to another program, 6–36
UNTIL phrase, 6–107
using ALTER statement, 6–35
using GO statement, 6–81
transfers of data
displaying multiple lines, 6–74
low volume, using DISPLAY statement, 6–74
STRING functions, 6–150
STRING statement, 6–149
with DISPLAY
sequence of transfer with more than one operand, 6–74
TRANSLATE option
CALL INTRINSIC statement, 6–51
USING phrase, 6–51
transmission of data
eligibility for in RECEIVE, 6–118
trouble-shooting
documenting problems, 1–2
truncating
results of the DIVIDE statement, 6–77
truncation
during moves, 6–96
of integer positions, 6–26
using ROUNDED phrase, 6–26
truth tests
in the sign condition, 6–20
truth values
conditional statements, 6–10
defined for complex conditions, 6–20
determining
evaluating complex conditions, 6–24
logical operators in complex conditions, 6–20
negating simple conditions, 6–21
of a condition, 6–15

U

unblocked file
definition, 5–20
uninitialized pointer dereference error
description of, B–2
UNIQUENAME file attribute
changing, 7–20
function, 7–20
Unisys
extensions
notation in manual, 1–3
UNITNO file attribute
function, 7–20
restrictions, 7–20
UNLOCK option
CALL INTRINSIC statement, 6–46
UNLOCK statement
Conversion Aid changes, F–35
unpredictable results
from level 2 IPC, 9–16
unreferenced data blocks
COBOL compiler final code file, D–2
UNSTRING
program example, E–1
UNSTRING statement
ALL phrase, effect, 6–154
as a conditional statement, 6–10
data transfers, rules, 6–156
DELIMITED BY phrase, 6–154
DELIMITER IN and COUNT IN phrases, 6–155
DELIMITER IN phrase, 6–157
example reference, 6–157
figurative constant as delimiter, 6–154
format 1, 6–153
format 2, 6–153
function, 6–153
initializing data items, 6–155
ON OVERFLOW phrase, 6–156
overflow condition, 6–155
PICTURE character strings, 6–154
POINTER phrase, 6–155
recognition of delimiters, 6–154
restriction on identifiers and level 88 entries, 6–153
sending area, 6–154
subscripting and indexing for identifiers, 6–154
syntax elements, 6–153
TALLYING phrase, 6–155
UP BY
function in SET statement, 6–137

UPON phrase
in DISPLAY statement, 6–74
uppercase words
notation
discussion, 1–3
USAGE clause
character representation, 5–68
DISPLAY data items, 5–68
format, 5–67
function, 5–67
level numbers, 5–68
representation of data items, 5–68
restrictions, 5–67
writing at group level, 5–67
USAGE IS DISPLAY
references in INSPECT statement, 6–85
using CODE-SET clause, 5–22
USAGE IS DISPLAY clause
function, 5–67
USAGE IS INDEX
SEARCH statement, 6–128
USAGE IS INDEX clause
elementary items, 5–68
group item, 5–68
index data items, 5–13
restrictions, 5–68
USASCII
bit representation example, C–1
column number specification, C–1
high-order bit specification, C–1
row number specification, C–1
USASCII X3.4–1963
code set differences, C–1
table of code sets, C–2
USASCII X3.4–1967
code set differences, C–1
table of code sets, C–3
USE AFTER STANDARD BEGINNING statement
using, 6–166
USE AFTER STANDARD ENDING statement
using, 6–166
USE AFTER STANDARD EXCEPTION declarative
use with WRITE statement, 6–174
USE AFTER STANDARD EXCEPTION procedure
with shared files, 8–24
USE BEFORE STANDARD BEGINNING statement
using, 6–166
USE BEFORE STANDARD ENDING statement
using, 6–166
USE FOR DEBUGGING section
execution of statements, D–3
USE FOR DEBUGGING statement
ALL PROCEDURES phrase, 6–161
format 1, 6–158
limitations on elements, 6–159
when debugging section is executed, 6–161
when executed, 4–2, 6–161
USE ON PROGRAM-FAULT statement
function, 6–165
USE procedure
control after execution, 6–164
defining conditions for execution, 6–164
execution of more than one, 6–164
handle parity error, example, E–3
in the START statement, 6–144
references to nondeclarative procedures, 6–164, 6–166
restrictions, 6–164
restrictions on the use of statements, 6–166
tape label handling, example, E–16
USE statement
ALL PROCEDURES phrase, 6–161
ALL REFERENCES phrase, 6–160
automatic execution of debugging section, 6–159
Conversion Aid changes, F–35
debugging section
execution conditions, 6–162
DEBUG-ITEM register, 6–161
ERROR and EXCEPTION, 6–164
execution conditions, 6–163
execution of more than one USE procedure, 6–164
format 1, 6–158
format 2, 6–158
explanation of function, 6–164
format 3, 6–158
overview and function, 6–165
format 4, 6–159
overview and function, 6–165

USE statement (continued)
function, 6–158
imperative verbs, debugging, 6–160
monitoring data items, 6–160
monitoring files, 6–161
monitoring procedures, 6–161
OCCURS clause, 6–160
organization of and access to files referenced, 6–165
placement and formatting requirements, 6–164
placement of debugging sections, 6–159
placement of statements, restrictions, 6–159
references in ALTER statement, 6–163
references in SORT or MERGE statement, 6–163
syntax elements, 6–159
USE FOR DEBUGGING statement
limitations, 6–159
user items
monitored by associated debugging section, D–3
user-defined options (*See* dollar options, user-defined)
user-defined words
assigning to implementer name, 2–5
character set, 2–4
list, 2–4
restrictions, 2–4
USING phrase
CALL statement , 5–42
defining operands, 6–37
definition of operands, 6–2
inclusion in called program, 6–37
MERGE statement
maximum file-names specified, 6–90
referencing files that are not open, 6–98
restriction on specifying data-names, 6–3
SORT statement, 6–142
SPOMESSAGE option, 6–48
use in Procedure Division, 6–2
WAIT statement, 6–168
with GETMCP option, 6–49
with TRANSLATE option, 6–51
with ZIPSPO option, 6–49

V

VALUE clause
condition-name entries, 5–69
Conversion Aid changes, F–20
Data Division
function, 5–69
File Section
rules for use, 5–69
format, 5–69
literals with condition-names, 5–69
restrictions, 5–70
rules
alphabetic data items, 5–70
alphanumeric data items, 5–70
alphanumeric-edited data items, 5–70
numeric data items, 5–70
numeric-edited data items, 5–70
setting initial values, 5–71
using with hexadecimal literals, 2–13
when USAGE is specified, 5–70
Working-Storage
rules for use, 5–69
VALUE command
setting switches, 4–7
VALUE OF clause
coding example, 5–32
example, 7–3
function, 5–32
general format, 5–32
restrictions on use of attributes in, 5–32
SECURITYFAMILY attribute, 7–17
SECURITYGUARD attribute, 7–17
SECURITYTYPE attribute, 7–17
SECURITYUSE attribute, 7–18
SHARED attribute, 7–18
specifying file attributes, 7–3
variable data items
restriction on VALUE, 5–70
variable length records (*See also* RECORD CONTAINS clause), 5–31
block size
Determining, 5–31
declaring, 5–21
determining block size, 5–20
READ with INTO phrase, 6–114

variable length records (continued)
tape
with SORT, 6–140
using INTO phrase with RETURN, 6–122
VARYING phrase
SEARCH statement, 6–127
VOID nnnnnn option
function and conditions, 9–21
primary source file, 9–21
voided images
listing in the output, 9–16
VOLUMEINDEX file attribute
function, 7–20

W

WAIT statement
ATTRIBUTE CHANGEEVENT, 6–168
ATTRIBUTE INPUTEVENT clause, 6–169
complex
use in testing, 6–168
Conversion Aid changes, F–44
CRCR-INPUT, 6–168
determining length for program suspension, 6–167
equivalence of READ-OK and INPUTEVENT, 7–25
equivalence of WRITE-OK and OUTPUTEVENT, 7–27
example, 6–169
finding event that caused WAIT to end, 6–169
format, 6–167
function, 6–167
ODT-INPUT-PRESENT is specified, 6–168
program suspension time, 6–167
READ-OK and ATTRIBUTE INPUTEVENT, 6–168
results of specifying GIVING, 6–168
results of specifying STOQ-INPUT, 6–168
specifying the subport, 6–168
STOQUE parameter block, 6–168
syntax elements, 6–167
USING clause, 6–168
WAKE-UP-CAUSE, 6–169
WRITE-OK and ATTRIBUTE OUTPUTEVENT, 6–168
WAKE-UP-CAUSE clause
contents, when WAIT ends, 6–169
WARNFATAL option
treatment of warnings, 9–21
warning messages
compiler, B–2
federal level, B–17
on output listing, 9–22
specification of FEDLEVEL, 9–13
suppressing listing, 9–21
treatment of, 9–21
WARNSUPR option
function, 9–21
WHEN phrases
SEARCH statement, 6–128, 6–130
WITH DEBUGGING MODE clause
compile-time switch, D–3
debug module support, D–3
effect on debugging lines, 1–10
function during compilation, 4–2
WITH NO WAIT phrase
in WRITE statement, 6–175
READ statement, port files, 6–116
using for port files in CLOSE, 6–62
word-alignment
of record items, 5–36
words
COBOL, 2–3
restrictions, 2–3
syntax
definition, 1–3
user-defined, 2–4
work files
compiler, 9–2
effect of WRKP MCP option, 9–6
memory requirements, 9–24
workarea access files
offset greater than BLOCKSIZE, 7–10
Working-Storage
coding example, 5–71
coding tables, 5–10
Conversion Aid changes, F–19
data items, non-contiguous
level number 77, 5–45
discussion, 5–71
grouping data elements and constants, 5–71
initial values of data items, 5–71
logical record concept, 8–1

Working-Storage (continued)
non-contiguous
description, 5–71
format of 77-level, 5–71
grouping constants, 5–71
grouping items, 5–71
placement of a data-name in the, 5–41
placement of a FILLER keyword in the, 5–41
purpose, 5–1
required values of level numbers, 5–45
resident segment area, D–2
setting initial values, 5–71
WRITE BCT
discussion, 6–176
WRITE statement
ADVANCING phrase, 6–172
as a conditional statement, 6–10
changes to LINAGE-COUNTER during execution, 5–29
Conversion Aid changes, F–36, F–44
creation of duplicated values, 6–116
current record pointer, 6–172
efficiency
AFTER ADVANCING phrase, 6–172
BEFORE ADVANCING phrase, 6–172
end-of-reel condition, 6–174
equivalence to MOVE TO, 6–172
execution and record release, 6–171
file status data item, 6–172
FOOTING clause of LINAGE, 6–173
format 1, 6–170
format 2, 6–170
format considerations
ADVANCING phrase, 6–172
END-OF-PAGE phrase, 6–172, 6–173
function, 6–170
indexed files
write beyond boundaries, 6–175
INVALID KEY condition
effect, 6–172
indexed file, 6–171
relative file, 6–171
sequential file, 6–171
LINAGE clause, 6–173
hardware devices, 6–173
maximum record size, 6–172
mnemonic-name
where defined, 6–174
OPEN EXTEND statement, 6–100
page overflow
FOOTING clause, 6–173
port files
write beyond boundaries, 6–175
record-name, explanation, 6–170
record-name references, restrictions, 6–172
relative files
write beyond boundaries, 6–174
updating ACTUAL KEY data item, 6–171
USE AFTER STANDARD EXCEPTION declarative, 6–174
with FROM, 6–172
WITH NO WAIT phrase, 6–175
writing beyond boundaries, 6–174
WRITELOCK option
CALL INTRINSIC statement, 6–46
restrictions, 6–46
using ON OVERFLOW or USE, 6–46
WRITE-OK
function in WAIT statement, 6–168
WRITE-READ statement
Conversion Aid changes, F–44
WRITE-READ-TRANS
Conversion Aid changes, F–45
WRITE-TRANS-READ
Conversion Aid changes, F–45
WRKP MCP option
effect on compiler work files, 9–6

X

XREF option
function, 9–21

Y

YOURNAME attribute
equivalence to MYNAME, 7–27
using null attribute, 7–23
YOURUSERCODE attribute
function, 7–28
using null attribute, 7–23

Z

ZIP option
  CALL SYSTEM statement, 6–56
  cautionary note, 6–57
  examples, 6–56
  using for MCP control functions, 6–56
  using for programmatic scheduling, 6–56
ZIP statement
  Conversion Aid changes, F–36
ZIPSPO option
  CALL INTRINSIC statement, 6–49
  commands with which you cannot use ZIPSPO, 6–49
  GIVING phrase, 6–50
  USING phrase, 6–49

Special Characters

...
  function in code, 1–5
.
  using periods in code, 1–5
$
  function in compiler control, 1–11
$COPYBEGIN option
  as delimiter within library programs, 6–71
  in COPY statement, 6–72
$COPYEND option
  as a delimiter within library programs, 6–71
  in COPY statement, 6–72
$DMSICM
  restriction on HIGH MEMORY clause, 4–18
$DMSICM option
  restriction on the use of the HIGH MEMORY clause, 5–42
$ICM option
  creating an ICM, 6–53
  with CALL statement, 6–36
$LEV21PC option
  using with CANCEL statement, 6–58
  with CALL statement, 6–36
$MAP compiler option
  accessing greater than 1 MD of data, 5–42
  list of information included in summary from, 5–42
$MERGE option
  primary file, 9–7
$NEW option
  using in program output, 9–22
$OPTIMIZE option
  effect on size errors, 6–27
$POP option
  function with Boolean options, 9–10
$POP statement
  syntax of the, 9–10
$SET ICM record
  using with EXIT PROGRAM statement, 6–80
*
  notation
    purpose, 1–8
@
  as a delimiter, 2–13
"
  using as separator, 2–2
,
  using as separator in code, 1–5
;
  using as separator in code, 1–5
[]
  using brackets in code, 1–4
{}
  using braces in code, 1–4
-
  using hyphen in indicator area, 1–9
/
  using stroke in indicator area, 1–8
>
  function in arithmetic expressions, 6–15
  using greater-than sign in code, 1–5
<
  function in arithmetic expressions, 6–15
  using less-than sign in code, 1–5
+
  as editing sign control symbol, 5–53
  using plus sign in code, 1–5
—
  as editing sign control symbol, 5–53
  using dash in code, 1–5
=
  function in arithmetic expressions, 6–15
  using equal-to sign in code, 1–5

==
  as pseudotext delimiters, 2–3
& option control record
  CARDIN, F–3
  CARDOUT, F–3
  coding rules, F–3
  COPY, F–5
  default values, F–2
  DISKIN, F–3
  DISKOUT, F–3
  error conditions, F–6
  flagging errors, F–3
  inclusion of input and list option, F–3
  input options, F–3
  list options, F–3
  LISTNEW, F–3
  LISTOLD, F–3
  LISTUN, F–3
  NORESEQ
    remaining options, F–4
  OBJECT-COMPUTER name
    remaining options, F–4
  output options, F–3
  PACKIN, F–3
  PACKOUT, F–3
  purpose, F–2
  remaining options, F–4
  RESEQ option, F–4
  SWITCH, F–5
  SYNTAX, F–4
  TAPEIN, F–3
  TAPEOUT, F–3
& Record Errors
  conversion errormessages, F–49